GEOPROGRAMMER™

ASSEMBLY LANGUAGE ENVIRONMENT FOR USE WITH GEOS™

```
;****    Super Draw    ****
.include macroFile
.include constants

                .psect    StartAc
ProgStart:      LoadW     r0, Gra
                jsr       Graphi
                LoadW     r0, Mai
                jsr       DoMer
                rts

BrushIcon:
```

FOR THE COMMODORE 64, 64c AND 128 COMPUTERS.

BERKELEY Softworks

geoProgrammer
User's Manual

Berkeley Softworks
2150 Shattuck Avenue
Berkeley, California 94704

Update Policy

To participate in Berkeley Softworks' update service, fill out and return the GEOS Registration Card found at the back of the manual. Registered users will be sent notices outlining the procedure for obtaining updates and revisions.

License and Limited Warranty

This manual and software are subject to all the terms of the accompanying Software License Agreement. Except for the limited warranty on the diskettes which is described in the Software License Agreement, **THE SOFTWARE AND ACCOMPANYING MATERIALS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.**

SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

IN NO EVENT WILL BERKELEY SOFTWORKS, INC. BE LIABLE FOR ANY DAMAGES, INCLUDING LOSS OF DATA, LOST PROFITS, COST OF COVER OR OTHER SPECIAL, INCIDENTAL, CONSEQUENTIAL OR INDIRECT DAMAGES ARISING FROM THE USE OF THE SOFTWARE OR ACCOMPANYING MATERIALS, HOWEVER CAUSED ON ANY THEORY OF LIABILITY. THIS LIMITATION WILL APPLY EVEN IF BERKELEY SOFTWORKS, INC. OR AN AUTHORIZED DEALER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. YOU ACKNOWLEDGE THAT THE LICENSE FEE REFLECTS THIS ALLOCATION OF RISK. SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

Manual written by Matthew G. Loveless

geoAssembler and geoLinker designed by Ted H. Kim
geoDebugger designed by Eric. E. Del Sesto
Project Manager: Eric E. Del Sesto

Printed 10/87

How to Get Help

We hope you will find geoProgrammer the ideal environment for developing GEOS applications and that this manual provides you with the answers to any questions you may have about using geoAssembler, geoLinker, or geoDebugger. However, if you do run across a problem that is not answered by this manual, there are several ways to obtain additional help.

QuantumLink

The fastest and most recommended way to obtain information about GEOS and GEOS applications such as geoProgrammer is through the QuantumLink telecommmunications network. QuantumLink (Q-link) is an online service network designed for Commodore users.

Berkeley Softworks provides Customer Service message boards along with a Programming and Technical Information board in the Commodore Software Showcase section of QuantumLink. Through these message boards, GEOS users and developers can generally receive the most timely help and information. In addition, you will have access to programs, products, and example source code from Berkeley Softworks which are offered through QuantumLink, many of them free of charge.

For more information on QuantumLink, call (800) 392-8200 from the United States. From Canada, call (703) 883-0788.

Telephone Support

Berkeley Softworks provides customer service by telephone, but, as the lines are often busy, it is recommended that you only call as a last resort. Additionally, our Customer Service department is not trained in answering detailed technical questions. Please submit such questions to our technical support staff via QuantumLink or U.S. mail. The Berkeley Softworks Customer Service telephone number is (415) 644-0890. Call between 9 a.m. and 5 p.m. Pacific Time.

Mail Support

If you mail your questions to the address printed in the back pages of this manual, Berkeley Softworks will answer your correspondence promptly. If you have a general question about GEOS or your geoProgrammer applications, send it attention: Customer Support; if you have a technical question about developing GEOS applications, send it attention: Technical Support.

User's Groups

In addition to Berkeley Softworks' official support, some of the most useful information comes from your local Commodore User's group. Often they will offer classes on 6502 assembly language and sessions with experienced GEOS programmers.

Table of Contents

(Chapter 4, cont.)

Chapter 5
geoAssembler Reference and Advanced Topics

Chapter 6
geoLinker Reference

Chapter 7
geoDebugger Usage and Tutorial

Chapter 8
Super-debugger Reference

Chapter 9
Mini-debugger Reference

(Chapter 9, cont.)

Appendices

Glossary

Index

Chapter 1: Introduction to 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 Commodore environment.

The geoProgrammer development system consists of three major components:

geoAssembler

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

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

geoLinker

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

geoDebugger

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.

Your geoProgrammer disk also has two sample applications which you can use as models for your own programs. In fact, we encourage you to copy the files and build upon them, using them as the basis for your applications.

You can also use the library of GEOS equate and macro files on the disk, making your source code easier to read and understand, as well as supporting (and extending) the standard in *The Official GEOS Programmer's Reference Guide*.

Using geoProgrammer with Other GEOS Based Programs

Since geoProgrammer is GEOS compatible, you can use it with other GEOS based programs.

geoWrite

Create geoAssembler source files and linker command files in your geoWrite word processor; include graphic and icon images from geoPaint and the Icon Editor directly into your source code; examine error files, symbol lists. geoWrite is included with the GEOS operating system.

geoPaint

Develop graphic images and icons for your applications with your geoPaint paint program. geoPaint is included with the GEOS operating system.

Icon Editor

Create and edit icon images for your applications with the GEOS Icon Editor. The Icon Editor is included with DESKPACK1. The forthcoming version 2.0 will allow photo scrap cut and paste operations.

How to Use This Manual

geoProgrammer was designed with the serious programmer in mind. It is therefore a sophisticated product. This does not mean it is hard to use, only that it must be approached in the proper way, with the proper prerequisites.

This manual will not show you how to use the GEOS deskTop; for that you'll have to refer to your GEOS User's Guide. Nor will it teach you 6502 assembly language; for that you'll have to refer to a good book on the subject. Finally, it will not show you how to program under the GEOS environment; that is the job of *The Official GEOS Programmer's Reference Guide*. However, this manual will attempt to bridge the gap between these other resources, thereby flattening an otherwise steep learning curve.

But the experienced programmer will not feel encumbered by this — many of the introductory chapters can be skimmed quickly before moving directly into the reference sections.

The manual is organized as follows:

Chapter 1 and Chapter 2 contain important information and procedures you should read and follow before you begin working with geoProgrammer. Chapter 1 gives you a general overview of the geoProgrammer system and this manual. Chapter 2 contains information on the equipment you need and the installation procedures you must follow in order to begin working.

Chapter 3 overviews the geoProgrammer development environment. It explains how geoAssembler, geoLinker, and geoDebugger interact, in addition to describing 6502 assembly language, the GEOS environment, and the application development cycle.

Chapter 4 explains the general use of geoAssembler and geoLinker. It describes how to create geoAssembler source code, assemble it, and finally link it into a runnable application. This chapter does not exhaustively cover the assembler and linker.

Chapter 5 is a reference chapter, covering all aspects of geoAssembler, from labels to expressions to macros. The chapter is designed to be both informative and convenient — providing quick and easy access to a breakdown of the assembler's features.

Chapter 6 is a reference chapter for geoLinker, covering all aspects of the link command file, and linker directives.

Chapter 7 overviews geoDebugger by introducing its major features and taking the reader through a brief tutorial session.

Chapter 8 is a complete reference for geoDebugger commands available in the Super-debugger. This debugger requires a ram-expansion unit.

Chapter 9 is a complete reference for geoDebugger commands available in the Mini-debugger.

Finally, the manual contains a number of appendices with useful information, as well as a comphrensive index and glossary.

We hope this manual helps you get the most out of your geoProgrammer development environment. We welcome comments and suggestions about the manual. Please send them to:

Berkeley Softworks
Attn: Documentation Department
2150 Shattuck Avenue
Berkeley, CA 94704

Conventions Used in This Manual

When important terms are first introduced, they are printed in *italics* to set them apart from the regular text. Many of these terms are further defined in the glossary at the end of this manual.

Paragraphs marked IMPORTANT, NOTE, and HINT appear throughout the manual. IMPORTANT alerts you to potential problems and suggest ways to avoid them. NOTE points out other information relevent to the topic at hand. And, HINT offers useful hints and tips.

Letters or words enclosed in rectangular boxes represent keys on your Commodore keyboard. Some functions require that you press and hold one key (like SHIFT) and then press a second key. In these cases, the keys will be listed serially with a plus (+) sign between them.

Syntax Notation

The following conventions are used in the syntax descriptions in this manual:

addrexp	address expression — a valid expression which evaluates to an address in the Commodore's memory space.
zp-address	zero-page address — a valid expression which evalutes to a zero-page address ($00-$ff).
exp *expression*	a valid expression.
filename	a valid GEOS file name which does not contain any spaces, whether leading, trailing, or embedded.
string	a string of ASCII characters enclosed in double-quotes.

symbol	a valid geoProgrammer symbol.
[ ]	square brackets indicate an optional item which may appear zero or one times.
{ }	curly braces indicate an optional item which may appear zero or more times.
|	a vertical line indicates a choice and can be read as "or".

Chapter 2: Before You Begin

Before you can begin to use the geoProgrammer system, you must read and follow the instructions in this chapter. This chapter will describe the equipment you need and the proper system configuration, how to install your geoProgrammer system, how to make a backup copy of your geoProgrammer disk, and how to make work disks for use with geoProgrammer.

What You Need to Use geoProgrammer

geoProgrammer is a part of the GEOS family of products. GEOS (Graphic Environment Operating System) is the official operating system for the Commodore 64. As a part of the GEOS world, there are certain pieces of equipment (hardware) and computer programs (software) which you need in order to run geoProgrammer. Additional equipment such as a printer, a second disk drive, a RAM-expansion unit (REU) are not required but will improve the performance and utilty of geoProgrammer. The REU is especially recommended for use with the geoProgrammer application due to its ability to bring increased speed and memory capacity to the Commodore 64/128 computer system.

You must have the following hardware and software in order to run and work with geoProgrammer:

- A Commodore 64, 64c, or 128 computer. Your 128 must be running in 64 emulation mode.
- One Commodore disk drive (1541 or 1571).
- GEOS (Graphic Environment Operating System) software version 1.2 or later, including geoWrite. You can upgrade to version 1.3 of GEOS and geoWrite by sending $5 to Berkeley Softworks Customer Service at the address printed in the back of this manual.
- An input device such as a joystick or a mouse.
- The geoProgrammer package, which includes the program diskette and this manual.

- Several blank, formatted disks for backup and work disks.

The following optional equipment is recommended to take full advantage of the power and versatility of geoProgrammer. This equipment is not necessary to use geoProgrammer.

- A RAM-Expansion unit (REU), such as the Commodore 1764 or 1750. With an REU, the operating speed of geoAssembler and geoLinker (and other programs) is greatly increased. This speeds up the turnaround time on the development cycle, thereby improving your programming productivity. Also, geoDebugger is designed to take advantage of the 64K system space in an REU, allowing you to debug applications which use the entire available program space.

- A GEOS supported printer that is properly connected to your computer. This will allow you to print out your geoAssembler source code, your geoLinker command files, and any error files. A list of GEOS supported printers is included in your GEOS User's Guide.

- An interface card or geoPrint Cable if you are planning on using a non-Commodore compatible printer to print out your GEOS files. geoPrint Cable is a parallel printing cable that makes printing your GEOS files fast and easy.

- A second disk drive (1541 or 1571). With two disk drives you will be able to copy files and disks more easily. You will also be able to dedicate all of the disk space on one disk to your source code, while the disk in the other drive contains the geoProgrammer system.

- A proportional input device such as the Commodore 1351 mouse. A proportional input device makes getting around in the GEOS world fast and easy.

- Several blank, formatted DS/DD (Double-Sided/Double-Density) diskettes for making work disks.

The geoProgrammer Disk

Your geoProgrammer system is contained on two sides of a floppy disk. Side A is the top, label side, and side B is the opposite side. To access the files on side B, the disk must actually be removed, turned-over, and reinserted into the drive. When you make a backup copy of your geoProgrammer disk, you will need to use two disks, copying side A to one disk and side B to another.

Following are the contents of your geoProgrammer disk:

Side A

GEOASSEMBLER	The macro assembler.
GEOLINKER	The overlay linker.
GEODEBUGGER	The symbolic debugger.
geosSym	complete GEOS symbols include file (no comments).
geosMac	GEOS macros include file (no comments).
SamSeq	Sample sequential application, main source code.
SamSeqHdr	Sample sequential application header source code.
SamSeq.lnk	Sample sequential application link command file.
SamSeq.dbm	Sample sequential application debugger macro file.

Side B

geosConstants	GEOS constants include file (with comments).
geosMemoryMap	GEOS memory map include file (with comments).
geosRoutines	GEOS routines include file (with comments).
geosMacros	GEOS macro file (with comments).
SamVlirRes	Sample VLIR application resident code module.
SamVlirEdit	Sample VLIR appliction Edit menu overlay module.
SamVlirFile	Sample VLIR application File menu overlay module.
SamVlirEquates	Sample VLIR application internal equates.
SamVlirZP	Sample VLIR application zero page variables.
SamVlirHdr	Sample VLIR application header source file.
SamVlir.lnk	Sample VLIR application link command file.
SamDA	Sample desk accessory main source module.
SamDAHdr	Sample desk accessory header source file.

SamDA.lnk	Sample desk accessory link command file.
DISK COPY	Disk backup utility for one-drive systems.

Installing geoProgrammer

Your geoProgrammer disk must first be installed into your GEOS system before you use it. You only perform the installation procedure once, the first time you use geoProgrammer.

IMPORTANT: Be sure to install geoProgrammer using your own GEOS boot disk or the GEOS boot disk that will always be used with this geoProgrammer disk. Any copies of geoProgrammer must also be used with this same GEOS boot disk.

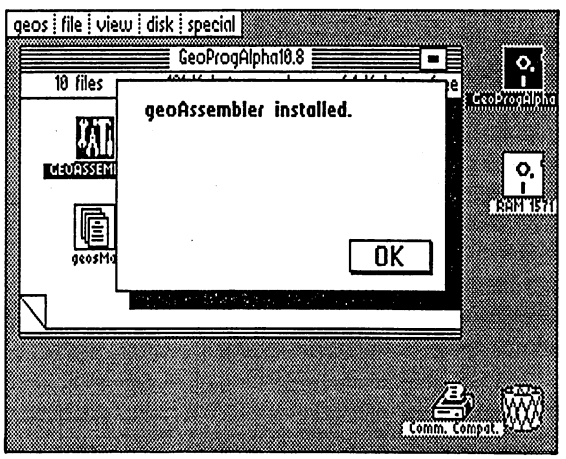
To install your geoProgrammer system, follow these steps:

1: Boot your copy of GEOS as described in your GEOS User's Manual.

2: Close your GEOS boot disk by clicking on the close icon in the upper-right corner of the window.

3: Put the geoProgrammer disk (label side, side A, up) into the disk drive and open it by clicking on the disk icon.

4: Open the file named geoAssembler by double-clicking on its icon or by selecting the geoAssembler icon (single-clicking on it) and choosing **open** from the **file** menu. The program will load and the following dialog box will appear:



5: Click on the **OK** icon to return to the deskTop.

6: Follow this same procedure (steps 4 and 5) for the geoLinker and geoDebugger files.

Your geoProgrammer disk is now completely installed. When you now run geoAssembler, geoLinker, or geoDebugger from the deskTop, rather than the installation procedure, you will be executing the actual program.

Making a Backup Copy of geoProgrammer

Before you actually start using geoProgrammer (but after you have installed it), you should make backup copies of your disk. In fact, once you have made a backup, you should store your original geoProgrammer disk away in a safe place. You should never use your original geoProgrammer disk for anything other than making backup copies.

With One Disk Drive

To make a backup copy of your geoProgrammer disk with only one disk drive, follow these steps:

1: Have two blank, formatted destination disks ready. Double-click on the DISK COPY utility program icon (located on side B of your geoProgrammer disk). The screen will turn blue. This is normal.

2: Follow the directions that appear on the screen to make a backup of side A of your geoProgrammer disk. The source disk is the disk you wish to copy *from* (your original geoProgrammer disk); the destination disk is the disk you wish to copy *to* (your blank backup disk). If you ran DISK COPY from side B of your geoProgrammer disk, you will need to turn it over to side A.

3: When the copy is finished, you will be asked if you wish to make another copy. Select yes and proceed with the copy, this time using side B of your geoProgrammer disk and the second blank, formatted destination disk.

With Two Disk Drives

GEOS must be set up to work with two disk drives as described in your GEOS User's Manual.

Follow these steps to make a backup copy of your geoProgrammer disk with two disk drives:

1: Place your original geoProgrammer disk in drive A, side A up, and a blank, formatted destination disk in drive B.

2: Select **copy** from the **disk** menu of the GEOS deskTop.

2: Follow the directions that appear on the screen to make a backup of side A of your geoProgrammer disk. The source disk is the disk you wish to copy *from* (your original geoProgrammer disk); the destination disk is the disk you wish to copy *to* (your blank backup disk).

3: When the copy is finished, you will be returned to the GEOS deskTop. Turn the geoProgrammer disk to side B and insert the second blank, formatted disk into the other drive. Now again select the **copy** from the **disk** menu to copy side B to the second disk.

These are the only *safe* ways to make copies of your geoProgrammer system disk.

IMPORTANT: Do not use the BACKUP program supplied with your GEOS disk. Only use the BACKUP program to make backup copies of your GEOS boot disk.

Making Work Disks

Once you have made one or more backup copies of your geoProgrammer disk, you will want to make work disks. A work disk is a disk you will use in your everyday development with geoProgrammer; you can make as many work disks as you like, and work disks can contain any combination of geoAssembler, geoLinker, geoDebugger, desk accessories, and your work files. In this way you can customize your work disks to suit your exact needs. For example, you might want one work disk with just geoAssembler, geoLinker, and your source files along with a second work disk with geoDebugger, your runnable application along with its debugger symbol file, and a file of debugger macros.

There are two ways to make a geoProgrammer work disk:

1: Use the DISK COPY program to make a work copy of Side A of your geoProgrammer disk onto a blank, formatted disk. With this new work disk, you can add or delete files as your needs demand.

2: Copy selected files individually from your geoProgrammer backup disk (and any other disk) to a blank, formatted work disk.

A work disk containing a selection of GEOS files might include the following:

geoAssembler	
geoLinker	
geoDebugger	
geoWrite	
roma	(font for geoWrite)
deskTop 1.3	(or later version)
printer driver	(the correct one for your printer)
geosSym	
geosMac	

This is a simple work disk configuration for geoProgrammer development. Depending on your needs, you can add additional files from other GEOS products and applications, such as:

- geoPaint, Graphics Grabber, and the Icon Editor so that you can add icons and images into your programs.

- desk accessories such as the Notepad, so that you can jot down memos and notes to yourself while you are working with geoProgrammer.

By having only the files that you need on your work disks, you allow for plenty of disk space for your geoAssembler source code. Make several customized work disks if you desire.

Chapter 3: Application Development

Chapter 3 overviews the geoProgrammer environment, beginning with a short introduction to assembly language, leading into the major elements of developing a GEOS application. Seasoned developers may want to merely skim this chapter, moving quickly to the reference portions of the manual.

After reading this chapter you should know:

- The difference between assembly language and machine language.
- The function of an assembler, linker, and debugger in the development cycle.
- The basic theory and practice behind GEOS program development.
- The general differences between sequential, VLIR, and desk accessory applications.

What is Assembly Language?

At the heart of every program you run — every paint program, word processor, computer language — lies 6502 *machine language*. Whenever your computer is on, the 6502† microprocessor inside is busy running through long lists of binary instructions (*binary* is the base-two number system most computers operate in; each digit is either 1 or 0, representing **on** or **off**). These binary instructions are machine language, the native language your 6502 understands. Machine language is the fastest, most elemental way of instructing your computer, and everything reduces to it. If you program in Commodore BASIC, for example, the BASIC interpreter must translate every instruction into a machine language equivalent, which may mean hundreds of binary instructions.

† The Commodore 64 actually uses a 6510 microprocessor, and the Commodore 128 uses an 8502 microprocessor. From a programming standpoint, these are identical to the original 6502, upon which they are based. In this manual, we will refer to this entire family of software-compatible microprocessors with the general term *6502*.

But while machine language is well-suited for computers to understand, most humans have trouble making sense out of a 11000101 or 00101100. Only the most self-punishing programmer would program directly in machine language. But that is why assemblers were developed. *Assemblers* allow programmers to design machine language applications using English abbreviations called *mnemonics*. "Mnemonic" comes from a greek word meaning memory and that is essentially what one is: a memory aid. Rather than cryptic strings of 1's and 0's, we are able to program with sensical words like JMP for *jump* and LDA for *load a*ccumulator. Assemblers will then translate these mnemonics into machine language instructions. This more-palatable way of programming is called *assembly language*.

Developing With geoProgrammer

Assembling, the process of converting assembly language *source code* into machine language, is only one step of the development cycle and only one third of your geoProgrammer development kit (geoProgrammer also includes a linker and a debugger).

geoAssembler

geoAssembler is a subset of an extremely powerful cross-assembler (microPORT), originally designed to run on larger, more sophisticated computers than the Commodore 64/128. In the conversion to the Commodore environment, most of the advanced functionality of microPORT development system has been preserved.

geoAssembler supports macro programming, conditional assembly, nested file inclusion, complex expression evaluation, and the standard 6502 mnemonic instruction set. In addition, your geoProgrammer disk contains a variety of equate and macro files which define commonly used variables, constants, and macros for the GEOS operating system. These files may be included with your own assemblies.

geoAssembler generates *relocatable object code*. This means that its output is not directly runnable, but must be first passed through geoLinker and resolved to an absolute address.

geoLinker

The most advanced aspect of the geoProgrammer system, and possibly the hardest to understand, is the *linker*. When you assemble a source file, geoAssembler does not produce a runnable program file. Instead, the assembler generates a .rel relocatable object file . This .rel file, as it stands, is not 6502 machine language; rather, it is in an intermediate form. This file must then be passed through the linker, which will generate a GEOS compatible, runnable file with the proper file header and icon information.

geoLinker can combine one or more .rel object files into an executable program. This allows you to split a large program across a number of source files, assembling these files independently and then linking all the resulting .rel files into one runnable program. Not only does this facilitate modular programming, it can also cut down on development time: if you make a change to an independent source code file, you need only reassemble that file and then relink with the already existing .rel files. Linking is appreciably faster than assembling.

The linker also allows you to create libraries of commonly used routines. Any time you need, say, string manipulations, you could link with a string.rel file you might have created during an earlier project. Building powerful libraries is one of the tricks to effective professional development — once you've programmed and debugged a generalized routine, you need never look at (or reassemble) it again.

geoDebugger

geoDebugger is the third leg of the geoProgrammer development system, and, at times, it may be the most indispensible. geoDebugger is a small program which co-resides with your GEOS application and facilitates the debugging process, allowing you to disassemble, modify, and trace the execution of your program. It is also a *symbolic* debugger, which means it will use labels, symbols, and equates from within your source code when displaying and operating on memory locations and program code.

The Development Cycle

geoProgrammer is a sophisticated development environment for GEOS applications — it encourages well-structrured programs, while lending itself, specifically, to efficient development under the GEOS environment. GEOS programs tend to be larger and more modular than traditional 6502 applications and demand the advanced features found in this package.

The Design Stage

The first step in any large project is to design the program. This usually means drawing up specs for the user-interface as well as puzzling out the organization, algorithms, and program structure. Under GEOS, it is especially important to design the user-interface early because the icon/windowing environment is so central to the development effort.

Event-driven Programs

GEOS applications are *event-driven*, which means that most of the time is spent waiting for *events*. An event can be the press of a key, the click of the mouse, or a timer going off. After your program initializes itself, it passes control to GEOS. When an event occurs, such as the user clicking on an icon, GEOS vectors transfer control to the appropriate routine in your program to handle the event. When the event has been serviced, control is again returned to GEOS to await the next event.

Coding

After the basic design, the program is developed in modules. This means that individual pieces, subroutines — almost small programs in themselves — are developed. The first to be written is usually the main module, the *initialization*, which is run when the application is first executed; the initialization code sets up the event vectors, initializes variables to their defaults, and draws the initial display.

geoAssembler source code is created with geoWrite. Although geoWrite is a word processor, it is also a powerful and familiar editing tool, and it lends itself well to this sort of application. As an added benefit: because geoWrite is a graphic word processor, you may include icon images (from geoPaint) directly into your source code; geoAssembler will convert the graphic images into compressed image data during assembly.

NOTE: geoWrite and geoPaint are not included on your geoProgrammer disk. They are included with the GEOS operating system.

Modules

Because geoProgrammer allows multiple .rel files to be linked into one application, each event routine can be relegated to its own source file and be assembled separately. Additionally, the geoProgrammer disk contains macro and equate files which may be included with your assembly. These files define macros, variables, and constants for the GEOS operating system. Using these files will make your programs easier to read as well as conform to the standards established in *The Official GEOS Programmer's Reference Guide*.

Assembling

Once a routine or source file is written, it may be assembled. The assembly process is simple: you merely invoke the assembler with the desired source code file and it does the rest of the work. The assembler reads in the source file and begins processing it. geoAssembler can create two types: a .rel linkable object file and a .err error file. The .rel file is linkable object code, and the error file is a geoWrite document which records any errors or messages in the assembly.

If there are errors in the assembly, usually caused by typing mistakes or the use of invalid instructions and addressing modes, they can be fixed at this time and the file reassembled. When all your source code files assemble without errors, you are ready to move on to the linking process.

Linking

Unlike the assembler, the linker uses a *command file*. The command file contains important information which tells the linker, among other things, the type of executable file to generate (sequential, VLIR, or Commodore), the file header to use, the proper load address, and the .rel files to include in the link. The linker reads in the .rel files, resolves all external references, and, if there are no errors, generates a runnable object file.

Debugging

Once you have gotten successfully through the assembly and link phases, you are ready to test the program. It is rare indeed when a program works correctly the first time; sometimes the icons aren't centered correctly, the menu items are misspelled, the screen erases itself, or perhaps the program halts entirely, locked forever in some endless loop. The process of tracking down and eliminating these "bugs" is called *debugging*, and debugging is one of the most frustrating (and rewarding) aspects of program development. Fortunately, the power of the geoDebugger makes the debugging process as painless as possible.

When you have discovered a bug, it's back to step one: you modify the source code to fix the problem, then reassemble, relink, and rerun. This whole circular process of program development is affectionately called the *assemble-link-crash-debug cycle.*

Application Types

GEOS supports three basic application types, all of which can be created with geoProgrammer:

- Sequential
- VLIR (Variable Length Indexed Record)
- Desk accessories

Sequential applications are the simplest and most straightforward type. Sequential files get their name from the way GEOS stores and accesses them on the disk: they appear as a contiguous block of data. When a sequential file application is executed, the entire program loads into memory. For most small and medium sized applications, those which can operate entirely in the free program area, a sequential format is sufficient. Only when programs get larger must you worry about other file formats.

VLIR applications are more sophisticated. Although the phrase "Variable Length Indexed Record" is a bit obscure, it is easy to understand the general concept. A VLIR application is never entirely in memory. Rather, only the necessary portions of the program, the parts which are in use, are loaded at any one time. When another part of the application is needed, it is simply loaded into a shared area of memory, overlaying routines or data which are no longer necessary. These portions of swappable code are called *overlay modules*. Using overlay modules, an extremely complex program, one with more machine code than could possibly fit in your Commodore computer, can be executed by loading in routines as they are needed. Designing a VLIR file application takes more forethought and effort than a sequential file application, but since the linker automates much of the drudgery, the process is certainly worth the effort for a more complex program.

Desk accessories are stored as sequential files and so are really not all that unique of an application type. The only difference in the file format is a special flag in the file's header and directory entry. You assemble and link desk accessories in the same way you would a sequential file, only setting the desk accessory flag in the header. Note, however, that desk accessories are designed differently than normal applications — they have special coding requirements and restrictions which are described in *The Official GEOS Programmer's Reference Guide*. geoProgrammer can also generate standard Commodore (non-GEOS) applications.

GEOS File Headers

Every GEOS file — whether a geoWrite document, a geoPaint picture, or an application you've created — has a corresponding 256-byte *header block* which is also stored on the disk. This header contains the icon image which appears on the deskTop, along with data describing the type of file, the starting address, and the loading address, among other information. When you design an application, you must also build a file header block. The file header block is a geoAssembler source file which generates the appropriate data; it is attached to your applications by geoLinker. For more information on building GEOS file headers, see **.header** in Chapter 5.

Chapter 4: geoAssembler & geoLinker Description and Usage

Chapter 4 describes the basic usage of the geoAssembler and geoLinker programs. It describes the syntax and format of geoAssembler source code, outlines the major features of the assembler, and demonstrates how to actually assemble a source code file. It also describes the general purpose of the linker and explains how to link files to produce a runnable program. This chapter *does not* cover aspects of the assembler and the linker in exhaustive detail (refer to Chapter 5 and Chapter 6 for more complete breakdowns). Rather, it serves to introduce you to the assembly-link process. If you are trying to learn assembly language, you should read this chapter along with the introductory chapters of a good 6502 assembly language book — many concepts which are only briefly touched upon here are covered in more detail by such books.

After reading this chapter you should know:

- The general format of geoAssembler source code, including line syntax and case-dependency.
- The following terms: mnemonic, opcode, operand, expression, directive, pseudo-op, label, equate, and macro.
- How to use geoWrite to create geoAssembler source files.
- The interaction of the assembler and the linker -- how they complement each other.
- How to run the assembler to generate relocatable object files. Also: you should understand the various files (.rel, .err) that the assembler generates.
- How geoLinker resolves cross-references and combines relocatable object files into a runnable program file.
- The purpose and function of the linker command file.
- How to operate the linker to generate a runnable program file. The various files generated by the linker (.err, .sym, .dbg, and the program file) will also be discussed.

How to Learn Assembly Language

We sometimes think of assembly language gurus as magical wizards who huddle around dusty old books and practice their arcane art with pentagrams and dragon's blood. But assembly language is not nearly as difficult or complex as its reputation might lead you to believe; in fact, it may be the very *simplicity* of assembly language which is hardest for most people to comprehend. Simple? Yes. Computers, at their most basic level, are very simple beasts — they are methodical, straightforward, and painfully simpleminded. Every task must be laid out explicitly and meticulously. This relentless demand for detail can stifle even the most intrepid learner.

In assembly language, for example, if you want to multiply five by six, you don't just say (as you might in BASIC) **5*6**. The 6502 has no multiply instruction. Instead, you must multiply five by six by *adding* five to itself six times! In this same way, if you want to search a string, open a disk file, or draw a line, you must use a routine which breaks the task down to a similar level of detail.

But because assembly language is the most basic form of programming, it is also the fastest, most flexible, and most compact. You can relish in the fact that your applications will be the best they possibly can.

As with most new skills, there are really just three essentials to learning (and eventually mastering) 6502 assembly language: patience, practice, and persistence. In addition, you should read a good book on 6502 assembly language (refer to Appendix D for reading recommendations).

6502 Source Code

MOS Technology developed the 6502 microprocessor in the mid-1970's and, along with it, a standard format for 6502 assembly language source code, including the popular three-letter mnemonics and addressing mode notation. All but the oldest books and magazine articles will assume this standard. geoAssembler implements a superset of the MOS Technology model; this means that geoAssembler will assemble most generic 6502 source code with very few changes.

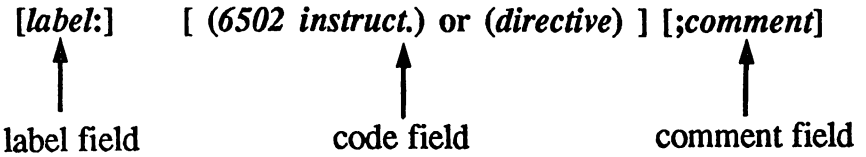
The following is a small 6502 subroutine which will assemble with geoAssembler:

```
;this line is a comment
.psect                                  ;assembler directive
start:      lda   #init_val             ;label defined
            asl   a                     ;a-mode addressing
            sta   3*buffer+2            ;expression
            cmp   #'c'                  ;ASCII character
            MoveW source,dest           ;macro with parameters
            rts                         ;implied addressing
Delay       =     20                    ;equate
.word       $50, delay, start           ;data definition
```

NOTE: The above code is designed to illustrate as many of the aspects of geoAssembler as possible. It is not intended to produce any useful results, nor to illustrate good coding practices.

General Syntax and Format

Assembly language source code follows a fairly simple set of rules. Source code is built up by lines and each source line (if it is not blank) is in the following general format:



Each field is optional, although when more than one is used, they must appear in the above order. In most cases, you will want to separate the fields with tabs, thereby making your source code neater and easier to read.

The *label field* may contain a label, which is an alphanumeric symbol or name of your choosing. It allows you to give meaningful names to your routines and variables. Although a label definition will usually begin at the left margin, you may insert as much *whitespace* (spaces or tabs) as you desire before defining a label. Labels must always end with a colon (:).

The *code field* may contain a 6502 instruction (mnemonic opcode and operand), an assembler directive (pseudo-op), or a macro invocation. The code field is usually indented one or two tab stops, but it may be surrounded by as much whitespace as desired. The code field is often subdivided into two separate fields: the *opcode field* and the *operand field.* The opcode field contains the instruction, macro, or directive and the operand field contains any necessary parameters, options, or 6502 operands. There must be at least one space or tab between the opcode field and the operand field.

The last field is the *comment field.* Comments are explanatory text or notes for describing your source code, analogous to the BASIC **REM** statement. A comment may appear anywhere on a line and must be preceded by a semicolon (;). *All* text following the semicolon is ignored by the assembler.

Case Dependency

geoAssembler takes advantage of both upper- and lower-case characters; it is a *case-dependent* or *case-significant* assembler. As a general rule, mnemonics, directives, and hexadecimal numbers may be typed in upper- or lower-case, or some mixture thereof, and geoAssembler will interpret them correctly: **lda #$Ab** is the same as **LDa #$aB.** However, with labels, equates, and macro names, the case is significant. That is: **label** is not the same as **LaBEL** or **Label.** Each unique occurrence of an upper- and lower-case combination is considered an entirely different symbol. For this reason

```
Loop:   inx
        lda     temp,x
        bne     Loop        ;correct
```

will assemble correctly. Whereas

```
        bne     loop        ;incorrrect!
```

(without the initial letter in the label capitalized), will generate an undefined label error.

Labels and Equates

Labels and equates allow you to use symbolic names within your assembly language source code. They make your programs easier to read, understand, and change, as well as automating much of the internal address calculations.

Labels and equates are similar in design and usage. They are both considered *symbols* and may be used in similar contexts. Symbols may be any combination of alphanumeric characters (remember: case is significant), but the first character must be a letter. You may also include the underline character (_) within a symbol name. Symbols can be as large as 20 characters, but the assembler will only consider the first eight; this means that **program_start** and **program_end** will appear the same to the assembler because the first eight characters (**program_**) are identical.

A *label* is a symbol which refers to a location within your actual program. This location can be either program code, initialized data, or variable space. A label is defined within the label field of a line and it is *always* followed by a colon. However, the colon is not considered part of the label name; the colon is the character which indicates to the assembler that it is a label definition. The absolute value (the actual memory location) of a label is resolved at link-time and this value is passed to the debugger in the symbol table.

An *equate* refers to an explicit definition of a symbol. You use the = or == directives to assign a value to the symbol. Equates can be addresses or constants.

Local Labels

Assemblers which do not implement *local labels* require the programmer to dream up sometimes hundreds of unique label names for even the most unimportant sections of code. The source code becomes cluttered with the likes of **loop1, loop2, loopxx4, lp,** and **lp002** which are not only confusing but unsightly. geoAssembler, fortunately, supports local labels. Local labels allow you to create labels which are *local* to a given routine or segment of code.

The *scope* of a local label, the range within which the label can be referenced, is limited to the area between any two regular (*global*) labels. A local label is a one to four digit number followed by a dollar-sign ($). Local labels do not need a trailing colon (:) — the dollar-sign is sufficient — but you may include one if you like. The following code segment illustrates the use of local labels.

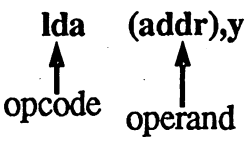
```
;**********************
;*** MOVE 256 BYTES ***
;**********************
Move_256:                   ;this is a global label
        ldy     #$00
1234$:                      ;this is a local label
        lda     (source),y
        sta     (dest),y
        iny
        bne     1234$
        rts
;
;******************************
;*** SET 256 BYTES TO NULL ***
;******************************
Kill_256:                   ;this is another global label
        ldy     #$00
        tya
1234$:                      ;this is a new local label
        sta     (source),y
        iny
        bne     1234$
        rts
```

Notice that although there are two occurances of the local label **1234$**, the scope of the first is limited to the area between **Move_256** and **Kill_256**. The scope of the second is limited to the area between **Kill_256** and the next (not shown) regular label. Note that the choice of **1234$** was arbitrary; it could just as easily have been **03$** or **771$**. Local labels can only be used as the destination of a branch instruction. They cannot, for example, be used in a mathematical expression or as the destination of a **jmp** instruction.

NOTE: At Berkeley Softworks, rather than use a **jmp** instruction, which won't work with local labels, we sometimes generate an unconditional branch — a branch which is always taken — with a **bra** (*br*anch *a*lways) macro. The macro expands to a **clv** followed by a **bvc**. This way, local labels can still be used as the destination. This macro is included in the sample macro file on your geoProgrammer disk.

Mnemonics, Opcodes, and Operands

6502 instructions consist of two distinct parts: the opcode and the operand.



The *opcode* is the actual 6502 instruction. In this case it is an **lda**, which stands for "*load accumulator.*" This three-letter abbreviation for the opcode is called a mnemonic. The difference between the mnemonic and the opcode is subtle: the mnemonic refers to the abbreviation for the instruction (e.g., **lda**), whereas the opcode *is* the actual instruction. The *operand* follows the opcode and is the address or value with which the opcode will "operate"; in the above example, the operand is the 6502's *indirect indexed* addressing mode.

Directives and Pseudo-ops

Directives are similar to 6502 instructions because they appear within the code field of a source line. However, directives (or *pseudo-ops* as they are often called) are not 6502 instructions. Rather, they instruct geoAssembler to perform some action. There are directives for assigning values to symbols (= and ==), incorporating other files into your source code (**.include**), macro definition (**.macro**, **.endm**), and conditional assembly (**.if**, **.else**, **.endif**), among others. Directives usually begin with a period to distinguish themselves from mnemonics and macros.

Comments

Comments add explanation to your source code. You should use them creatively and liberally wherever your program's actions are not immediately discernable. Comments begin with a semicolon (;) and extend to the end of a line. You may place a comment on a line all by itself, or you may place one at the end of any source code line.

Macros

A *macro* is the facility of geoAssembler which allows you, in essence, to create your own instructions and directives. You develop a group of source lines called the *macro definition* and give them a name. Whenever this *macro name* is subsequently used in your source code (within the code field), the assembler will replace it with the preassigned source lines, thereby *expanding* the macro. Macro expansion is not just trivial text replacement: macros expand dynamically at assembly time — you can pass up to six parameters to the macro at each *invocation* (use) and the macro can utilize those parameters in expressions, in conditional assembly, and even within additional macro calls.

Macros are extremely powerful and useful. For example, the 6502 has no move instruction. That is, it does not have the ability to move a byte or a word (two bytes) from one location to another with only one instruction. With the 6502, it takes two instructions: bytes must first be loaded into a register from the source address and then stored from the register to the destination address. This is a good candidate for a macro because it is a common operation. You might define a couple of macros: one called **MoveB** for move byte and one called **MoveW** for move word:

```
;MOVE BYTE MACRO
.macro    MoveB     source, dest    ;macro definition
          lda       source
          sta       dest
.endm

;MOVE WORD MACRO
.macro    MoveW     source, dest
          lda       source
          sta       dest
          lda       source+1
          sta       dest+1
.endm
```

If you then wanted to move something from **address1** to **address2**, you would need only say:

```
MoveB       address1, address2     ;move a byte

or

MoveW       address1, address2     ;move a word
```

where **address1** and **address2** are parameters which are passed to the macro.

Macros can be used for everthing from creating high-level control structures (like do...while, if...then, etc.) to abbreviating frequently used instruction sequences. Your geoProgrammer disk contains macro files for use with GEOS (refer to Appendix A for the more information on the included files).

Expressions

geoAssembler includes a comprehensive integer math package and expression evaluator. This means you may include mathematical and logical expressions in your source code which will be evaluated when the program is assembled. This makes it simple to create complex data tables and programs which dynamically adapt themselves based on a few initial equates. For example, you could do the following:

```
lda   #buf_size*10
sta   mem_rsrv + (module*4) + (fifo_siz/2)
```

Creating geoAssembler Source Code

You create geoAssembler source code with the geoWrite word processor included with your basic GEOS system. For instructions on operating geoWrite, consult the manual which came with the program. Because geoWrite was originally designed as a document processor and not a program text editor, there are a few things additional things to be aware of.

No Spaces in Filenames

The geoAssembler and geoLinker parser will not correctly interpret file names which contain spaces. To avoid any complications, do not place spaces (whether leading, trailing, or embedded) within the file names of your geoAssembler source code.

geoWrite Page Breaks

geoWrite is a page-oriented word processor. That is: it automatically divides your text into pages. At first this may seem odd, to break assembly source code into pages, but you will soon realize that it encourages good programming practices. A commonly accepted rule-of-thumb in programming is to have no routine that is longer than one page — the reasoning is based on the idea that any routine larger than a single page is needlessly complicated and should be broken into several smaller routines. With geoWrite breaking your source file into pages, you can better follow this rule. However, for the irreverent at heart, geoAssembler does not care about page breaks. If a routine crosses a page boundary, the assembler will treat it as a contiguous block of code.

Special Keystrokes

Many characters, such as the underscore and the tab, are common in geoAssembler source files. They are created in geoWrite as follows:

Tab		CONTROL + I
Underline	_	C= + -
V-bar	|	C= + ↑
Circumflex	^	↑
Tilde	~	C= + *

Tabs vs. Spaces

Get in the habit of using tabs (CONTROL I) to align your source code. Assembly language text lends itself nicely to vertical alignment, with opcodes, operands, and comments separated into columns. You can always use space characters instead of tabs (geoAssembler doesn't care), but it isn't recommended; space characters take up more space in memory and on disk, and they don't always line-up properly when using proportional text fonts.

Text Effects

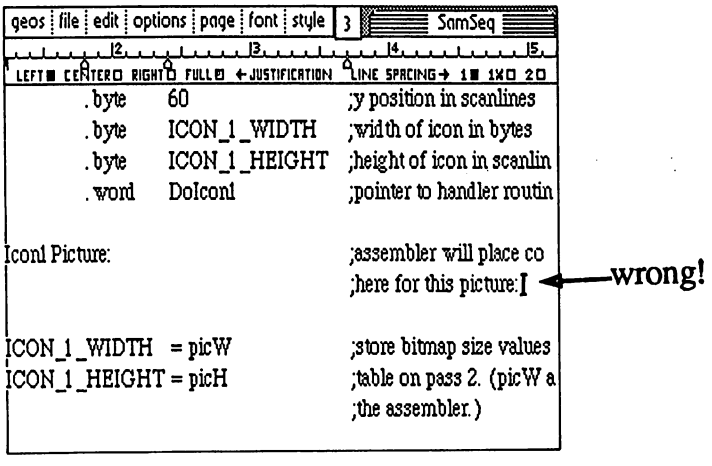
You may include special font and type effects, such as italics, directly into your geoAssembler source code. geoAssembler will ignore the special codes, converting all text into normal characters while assembling. This allows you to empahsize and highlight sections of your source code.

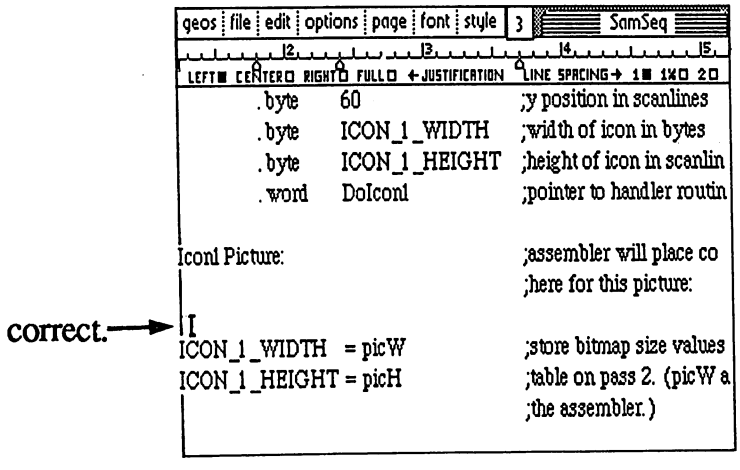
Including Icons (graphics) in Your Source File

One of the benefits of using a graphic word processor is that you are able to include blocks of bit-mapped graphics for icons and other images directly into your source code. geoAssembler will automatically convert these pictures into compacted bitmap data at assembly time. (For more information on GEOS compacted bitmap format, refer to *The GEOS Programmer's Reference Guide*.)

To insert a graphic image into your source code, place the geoWrite text cursor on a completely blank line in your source file and then paste the image as you would if you were including graphics in a regular document.

IMPORTANT: You *must* paste the graphic image into your source file with the text cursor on a completely blank line. If you do not, geoAssembler will ignore the data without reporting an error, even though it will appear correctly within the document.



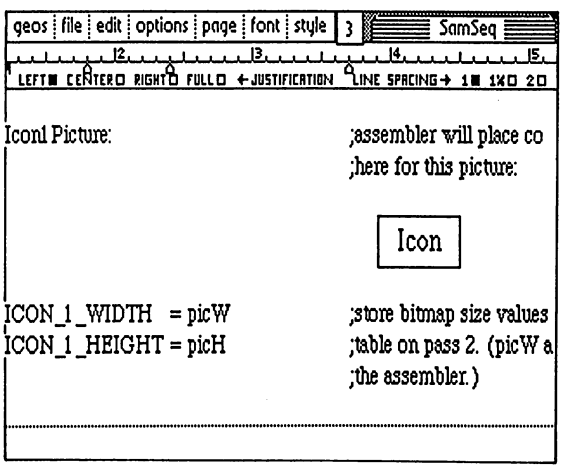


It is also a good idea to place an extra blank line at the end of each graphic image. You can do this by pressing [RETURN] immediately after pasting the image.

HINT: When cutting graphic images from geoPaint for inclusion in your source code, it is best to first turn color off, then move the image to the upper-left corner of the paint screen. This will ensure that the leftmost pixels are aligned on a card boundary (byte boundary). Any unused pixels (bits) on the right edge, up to the next byte, will be padded with zeros. You can also create icon images with version 2.0 of the Icon Editor.

PicH and PicW

For your convenience, geoAssembler maintains two internal variables which hold the size of the most recently defined graphic image: **picH** and **picW**. **picH** is the graphic image height in scanlines and **picW** is its width in bytes. These variables are redefined after each graphic image, so if you need the values, it is best to immediately assign them to a permanent equate. Here is an example:



Form more information on **picH** and **picW**, refer to "Internal Variables" in Chapter 5.

How the Assembler and Linker Relate

Most 6502 source code must be assembled to operate at at particular, absolute memory adddress. That is, if you assemble your source code to run at address $400, you cannot load it at $800 and expect it to run correctly. Most assemblers require that you explicitly declare the assembly address at the beginning of your source code in order to generate absolute code. geoAssembler, however, always generates relocatable object code-all labels and addresses are resolved at link-time relative to the other linked files. This allows you to assemble multiple source files without worrying about where each will begin and end; the address housekeeping is handled automatically by the linker.

NOTE: There is some confusion over the precise meaning of the terms relocatable and absolute. geoAssembler generates relocatable object code. This is code which is assembled at no specific address; at link time, the linker will determine the actual absolute address relative to a address given to the linker. Depending on the number and size of the .rel files, the absolute address will vary. Don't confuse relocatable with position-independent, which is something entirely different.

A typical, medium sixed application might have five separate source files which are eventually linked together to form the executable program file. Each of these source files shares a common ser of include files (files which are inserted in the assembly with the .include directive), and all are assembled into relocatable object files, designed to be asssigned an absolute address at the link stage.

Assembling

These source files must each, in turn, be assembled into .rel relocatable object files. One of the five source files is special. It is the header file, which contains the file icon image and other identifying data. All programs which run under GEOS must have a header. When you develop your own applications, you must create this header manually unless the default header serves your purposes well. The header is comprised primarily of **.byte** data statements and must be assembled just like the other source files.

An asssembly will generate either one or two files, both with the basic name of the source file but with a .rel or .err extender attached. The .rel file is the relocatable object code and the .err is the error file.

Linking

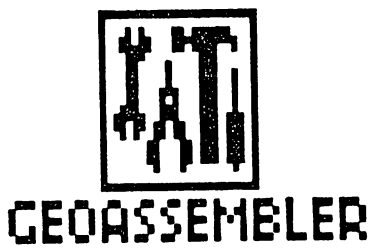
Once all the constituent .rel files have been created, they are ready for linking. You run the linker with a linker command file. The linker command file specifies the output file name, the header file name, the absolute addresses for program code and unitialized data segements, and the necessary .rel files to link. The linker will then run through the .rel files, resolving cross-references and relocatable addresses,and generate an executable program file.

Running geoAssembler

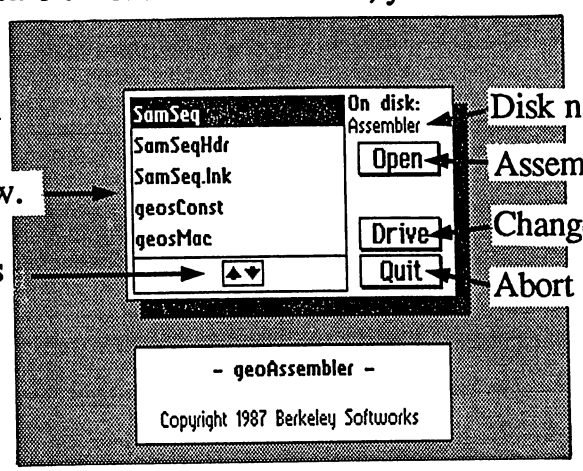
geoAssembler must be run from the GEOS deskTop. Please refer to your GEOS User's Manual if you have any questions relating to the operations of the deskTop.

To assemble a source file, follow these steps:

1: With your geoProgrammer work disk in the drive, double-click on the GEOASSEMBLER icon to run the assembler.



After the assembler loads and initializes, you should see the following dialog box:



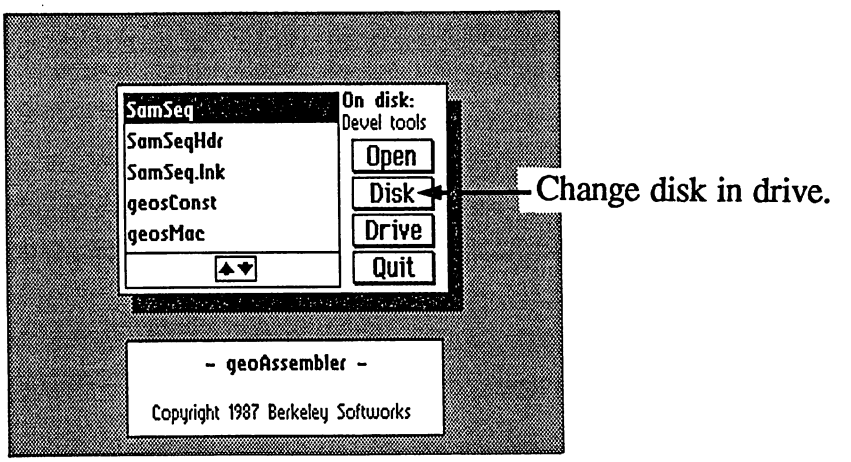
The contents of the current drive (the drive from which you ran geoAssembler) will appear in the directory window. If more items exist than can fit in the window, click on the scroll arrows to move through the directory.

IMPORTANT: Do not remove your geoAssembler work disk from the current drive until you return to the deskTop.

If you decide you do not want to do an assembly at this time, click on the **Quit** icon to abort and return to the deskTop.

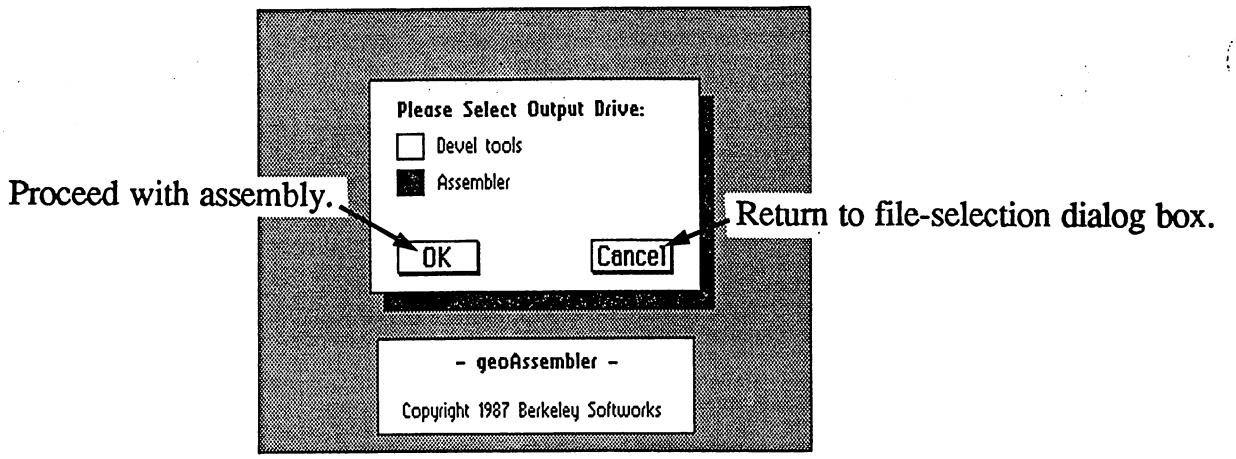
Select the file you want to assemble by clicking on the file name. Then click on the **Open** icon to initiate the assembly.

To assemble a file from a different drive (for example, a RAM Expansion Unit or a second floppy drive), click on the **Drive** icon; the directory of the other drive will be displayed in the directory window and a new icon labeled **Disk** will appear:



The **Disk** icon allows you to view the contents of a different disk. The **Disk** icon was absent from the original dialog box because you are not allowed to remove the disk which contains geoAssembler. To view the contents of a different disk, insert a new disk into the current drive and click on the **Disk** icon. The directory will be updated to show the contents of the new disk. The **Disk** icon will have no effect with a Ram Expansion Unit.

2: Once you have selected and opened a file you wish to assemble, you will see the following dialog box:



This dialog allows you to select the destination drive, the drive to which geoAssembler will write the output files (.rel and .err). It will default to the same drive as the source file. To select a different output drive, click on the box icon next to the disk's name. The icon will highlight. Click on the **OK** icon to proceed with the assembly, or click on the Cancel icon to return to the file-selection dialog box.

3: The screen will clear and geoAssembler will print a status message,indicating the progress of the assembly:

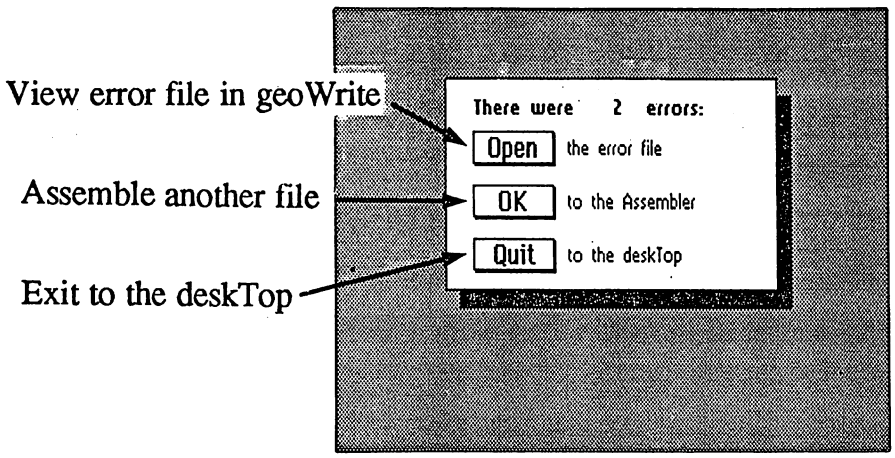
Assembling 0 .....

geoAssembler prints a period after every ten lines of source code. The number (which is zero when you begin) is a running error count and will increment after each error. This allows you to abort the assembly when you see a large number of errors. If the error count exceeds 99, geoAssembler will automatically abort the assembly.

The status message is printed at the bottom of the screen because geoAssembler temporarily uses the remainder of the screen memory area for the symbol and macro tables.

NOTE: You can abort an assembly by pressing the RUN/STOP key on the Commodore keyboard.

4: When the assembly is done, a dialog box describing the result of the assembly will appear:



Running geoLinker

Once you have assembled one or more .rel files from your assembly source code, you can use geoLinker to produce a runnable program file. geoLinker requires a linker command file such as the following:

```
.output   myprog
.header   myhead.rel
.seq
init.rel          ;initialization code
main.rel          ;main program code
```

This linker command file (created with geoWrite) will generate a runnable sequential program file called **myprog** with a header from **myhead.rel** and relocatable object code from **init.rel** and **main.rel**. The three .rel files were assembled previously. This is a very simple linker command file. More complex applications might require a full page of linker directives and object file names.

The Linker Command File (brief overview)

Linker command files are normal geoWrite text files except they follow a strict format and should have a .lnk file name extender. It consists mainly of linker directives and link file names. Comments may be added as they are in geoAssembler — on a line, anything following a semicolon (;) is ignored.

(For a complete breakdown of linker command files, refer to Chapter 6.)

Linking With geoLinker

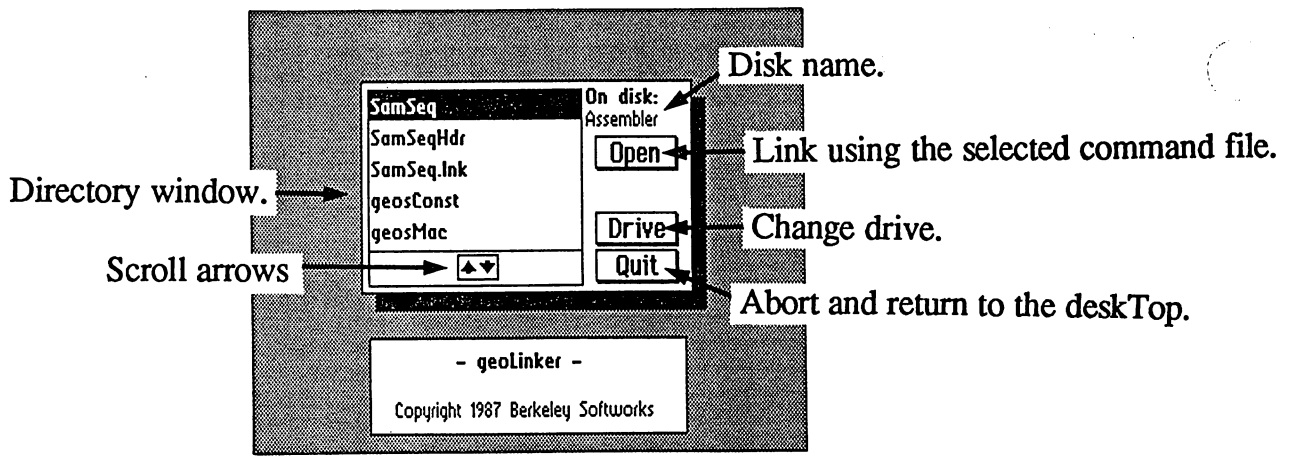
geoLinker, like geoAssembler, must be run from the GEOS deskTop. Please refer to your GEOS User's manual if you have any questions relating to the operation of the deskTop.

To create a runnable program file, you must first have created a linker command file and the proper, previously assembled, .rel files.

To actually perform a link, follow these steps:

1: With your geoLinker work disk in the drive, double-click on the GEOLINKER icon to run the linker.

After the linker loads and initializes, you should see the following dialog box:



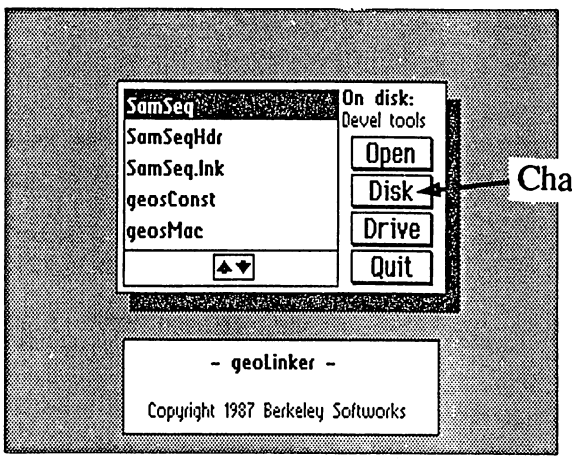
The contents of the current drive (the drive from which you ran geoLinker) will appear in the directory window. If more items exist than can fit in the window, click on the scroll arrows to move through the directory.

IMPORTANT: Do not remove your geoLinker work disk from the current drive until you return to the deskTop.

If you decide you do not want to do a link at this time, click on the **Quit** icon to abort and return to the deskTop.

Select the command file you want to link with by clicking on the file name. Then click on the **Open** icon to initiate the assembly.

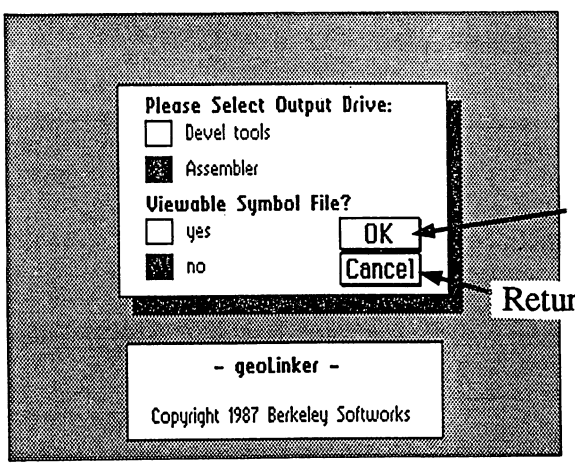
To use a command file on a different drive (for example, a RAM Expansion Unit or a second floppy drive), click on the **Drive** icon; the directory of the other drive will be displayed in the directory window and a new icon labeled **Disk** will appear:



Change disk in drive.

The **Disk** icon allows you to view the contents of a different disk. The Disk icon was absent from the original dialog box because you are not allowed to remove the disk which contains geoLinker. To view the contents of a different disk, insert a new disk into the current drive and click on the Disk icon. The directory will be updated to show the contents of the new disk. The **Disk** icon will have no effect with a Ram Expansion Unit.

2: Once you have selected and opened linker command file, you will see the following dialog box:



Proceed with link.

Return to file-selection dialog box

This dialog allows you to select the destination drive, the drive to which geoLinker will write the output files. It will default to the same drive as the source file. To select a different output drive, click on the box icon next to the disk's name. The icon will highlight.

At this point you can also select whether you want to generate a viewable symbol table. A viewable symbol table is a geoWrite file which contains a list of all the symbols which will be sent to the debugger. The viewable symbol table has a .sym extender, whereas the symbol table the debugger uses has a .dbg extender.

Click on the **OK** icon to proceed with the link, or click on the **Cancel** icon to return to the file-selection dialog box.

3: The screen will clear and geoLinker will print a status message, indicating the progress of the link:

Linking 0

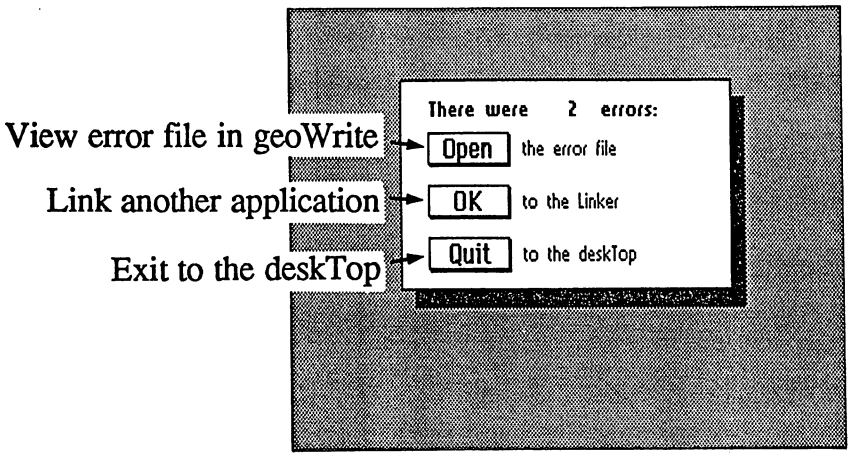
The number (which is zero when you begin) is a running error count and will increment after each error. The counter will stop after 99 errors, although any additional errors will still be written to the error file. You can abort the link when you see a large number of errors.

geoLinker also prints the file names of the .rel files as it processes them. When sorting the symbol table, geoLinker prints "sorting."

The status message is printed at the bottom of the screen because geoLinker temporarily uses the remainder of the screen memory area for the symbol tables.

NOTE: You can abort a link by pressing the RUN/STOP key on the Commodore keyboard.

4: When the linking is done, a dialog box describing the result of the link will appear:



Zero errors means a successful link. Anything else means the link was unsuccessful. At this point you can go directly to geoWrite and view the error file by selecting the **Open** icon; you can rerun the linker to link another file by selecting the **Ok** icon; or, you can return to the deskTop by selecting the **Quit** icon.

Successful Link

If geoLinker terminates without any errors, you will have a runnable program file on the selected destination drive. You may now test the program by running it from the deskTop or from within geoDebugger.

Unsuccessful Link

In the event of errors in a link, geoLinker will still generate an application file, and the associated .dbg debugger symbol file. At this point you will probably want to examine the .err file with geoWrite, fix the link errors, and relink, although you can choose to ignore the errors and attempt to run the file anyway. (For more information on the contents of the .err file, refer to Appendix E.)

Creating a Sample Application

Included on your geoProgrammer disk is a sample sequential application illustrating GEOS menus and icons. Everything you need to assemble and link the application is included on the disk.

To create the sample sequential application, follow these steps:

1: Copy the following files from your geoProgrammer backup disk to a disk which contains the deskTop and geoWrite, but is otherwise empty:

geoAssembler
geoLinker
geosSym
geosMac
SamSeq
SamSeqHdr
SamSeq.lnk

This will be your geoProgrammer work disk for the sample application.

2: Put your geoProgrammer backup disk away and open the work disk you just created. Run geoAssembler and assemble the following files:

SamSeq
SamSeqHdr

Two .rel relocatable object files will be created on the disk:

SamSeq.rel
SamSeqHdr.rel

3: Run geoLinker and select the **SamSeq.lnk** linker command file. geoLinker will relocate the **SamSeq.rel** file to an absolute address and attach the **SamSeqHdr** header to create the runnable application **SampleSeq** and a debugger symbol file **SampleSeq.dbg**. You can now run **SampleSeq** from the deskTop.

Later, in the geoDebugger chapter, we will use the **SampleSeq** application in a tutorial session with the debugger.

Chapter 5: geoAssembler Reference and Advanced Topics

Chapter 5 acts as a complete reference for geoAssembler source code format, including line syntax, assembly control, expressions, labels, directives, and macros. Although this is primarily a reference chapter, it would be a good idea to read it through completely at least once. For information on using geoAssembler from the GEOS deskTop, refer to "Running geoAssembler" in Chapter 4.

The Assembly Process

geoAssembler is a *two-pass* assembler. That means it processes the source code file twice in order to correctly resolve both forward and backward references. During both passes, geoAssembler maintains three independent counters which determine the placement of your object code: a *zsect* counter, a *psect* counter, and a *ramsect* counter. These counters refer to three distinct sections within the eventual application: zero-page, program code, and unitialized dataspace.

Zero Page (zsect)

The 6502 supports a special form of addressing called *zero-page* addressing. Zero-page (or page 0) refers to the first 256 bytes ($00-$ff) of memory; Instructions which use zero-page locations take up less space and operate significantly faster than their counterparts which use the remainder of the addressing space. Because geoAssembler takes special actions when it encounters zero-page variables on the first pass, zero-page variables must be defined before they are used, and they must be defined within a special section of your source code using the **.zsect** directive.

Program Code (psect)

Program code and initialized data are stored in the psect section. *Program code* refers to 6502 instructions and *initialized data* refers to icon images and data created with the **.byte** and **.word** directives. The absolute location of the psect section is determined at link time and is usually specified in the linker command file. You begin a psect section in your source code with the **.psect** directive.

Unitialized Data areas (ramsect)

The ramsect section maintains unitialized data areas of your program. Unitialized data definitions within your source code allow you to reserve memory space for your program's use with the **.block** directive. Ramsect areas take up no room in the program file generated by the linker; the space is established when the program is executed. geoAssembler allows you to specify an absolute starting address for the ramsect section at assembly-time, but if you supply no parameter in the **.ramsect** directive, the absolute address will be established at link-time.

Pass One and Pass Two

On the first pass through the source file, geoAssembler increments the three section counters and determines the values of all the symbols which are defined or equated in the source file. On the second pass local labels and forward references are resolved and any .rel or .err output files are generated.

Assembler Input

Lexical Analysis

geoAssembler evaluates the source file a line at a time. Each source line is in the following general format:

```
LABEL:    OPCODE:   OPERAND:      COMMENT:
Start:    lda       #$ff          ;load immediate   addressing
          sta       table,y       ;store indexed with y
          iny
reset2:   lda       (z_temp),y    ;load indirect indexed
77$:      rol       a             ;accumulator addressing
```

geoAssembler ignores blank lines and geoWrite text formatting codes. However, it will convert image data within your source files into compacted bitmap data at assembly-time.

For a more basic breakdown of geoAssembler source code format, refer to "General Syntax and Format" in Chapter 4.

Symbols

A symbol is a global label or an equate. A symbol must begin with an alpha character (**A-Z** or **a-z**), but the remaining characters can be numbers (**0-9**) or underline symbols (**_**). Case is significant within a symbol name. Symbols may contain as many as 20 characters, but geoAssembler only stores the first eight.

geoAssembler reserves some symbols for its own internal use. These include the upper- and lower-case **a**, **x**, and **y**, which are used for register mode addressing, the special graphic symbols **picH** and **picW**, and the **Pass1** flag. Also, although it is possible, it is not a good idea to use mnemonic names (such as **lda** or **rol**) as symbols.

HINT: Use descriptive names for variables, routines, and constants; avoid symbol names which could easily be confused, such as **pos1** and **posl** (the numeric "1" and the lower-case "l"); distinguish two related labels by their initial character rather than a trailing one — e.g., geoAssembler would interpret **position_X** and **position_Y** as the same symbol (because the first eight characters are identical), but not **X_position** and **Y_position.**

Equates

An equate is a symbol which is given an explicit value with either the = or == assembler directive. The only difference between the = directive and the == directive is that equates made with the double equal-sign are sent to the debugger, whereas those made with a single equal-sign are not. This allows you to avoid cluttering geoDebugger's symbol table with unneeded equates. Both types of equates, however, are still passed to the linker unless they are preceded by a **.noeqin** directive, which will limit their scope to the current assembly file.

Examples:

```
bitmask  ==   %01001110          ;will be sent to debugger
null     =    0                  ;will not go to debugger
S_flag   =    (%0100 & bitmask)  ;will not go, either
```

IMPORTANT: All equates must be resolvable on the first pass of the assembly. This means you cannot use any forward, relocatable, or external references in the definition of an equate; all symbols which are used in an equate definition must already be defined with an absolute address.

Labels

Labels are symbols which take the value of the current section counter. In a psect section for example, a label will be assigned the current value of the psect counter. Likewise, labels in the zsect section will take the current value of the zsect counter, and labels in the ramsect section will take the current value of the ramsect counter. All psect labels and most ramsect labels are relocatable — they are not given an absolute address until link-time.

Labels are defined by placing a symbol within the label field of a source line and following it with a colon (:). Note, however, that the colon is not actually part of the symbol's name — subsequent *uses* of the label must omit the colon.

Examples:

```
.zsect      $20          ;set initial zsect counter value
temp1:      .block 1     ;temp1 will equal $20
temp2:      .block 1     ;temp2 will equal $21
pointer:    .block 2     ;pointer will equal $22
Mem_free:   .block 4     ;Mem_free will equal $24

.psect                   ;linker will calc abs location
Start:
            ldy     temp2
            lda     X_table, y
            sta     temp1
            jsr     set_mouse
drop:       lda     (pointer), y
            sta     X_mouse
            iny
            lda     (pointer), y
            sta     Y_mouse
            rts

X_table:    .byte   mouse1, mouse2, mouse3, mouse4
            .byte   mouse5, mouse6, mouse7, mouse8

.ramsect                 ;let linker calc. abs location
X_mouse:    .block 1
Y_mouse:    .block 1
```

Note that these labels are *global labels* — they can be accessed from anywhere within the current assembly file and cross-referenced from other relocatable object files if they are passed to the linker. Labels which follow a **.noglbl** directive are not passed to the linker. This means that such labels are hidden from other modules at link-time; the *scope* of **.noglbl** labels is limited to the current assembly. The default is to send all global labels to the linker

Local Labels

Local labels consist of one to four numeric digits followed by a dollar-sign ($) in the form **nnnn$**, where **nnnn** is a one to four digit number. Local labels are only visible to code within the current *local region.* Local regions are delimited by successive global labels. When defining a local label, the colon (:) after the label is optional.

NOTE: Although local labels are made up of numeric characters, they are not in fact numbers — the one to four digits are treated as a text string. For this reason, **0071$, 071$,** and **71$** are all different local labels.

You can only use a local label as the destination of a branch instruction from within the same local region. Local labels are not passed to the linker and are not included in the symbol table. They are resolved on the second pass of the assembly.

Example:

```
routine:    lda     #$00        ;start of a local region
            ldy     #count
1$:         sta     bitmap, y   ;local label defined
            lda     Xmap, x
            cmp     #abort
            beq     86$         ;branch to local lbl    (forwards)
            inx
            dey
            bne     1$          ;branch to local lbl    (backwards)
86$:        rts                 ;another local label    defined
routine2:   lda     #$ff        ;next global; delimits local
                                 region
```

IMPORTANT: Avoid using large-value local labels such as **9999$** and **9988$** because the macro processor generates local labels counting backwards from **9999$**. You should have no problems with local labels less than **9000$**. For more information, refer to **.macro** later in this chapter.

6502 Opcodes and Operands

Opcodes

geoAssembler recognizes the full set of MOS Technology 6502 mnemonics. There are 56 in all, and they can be found in books describing 6502 assembly language.

Some 6502 assemblers support alternate mnemonics for various instructions, such as **bge** (*b*ranch on *g*reater than or *e*qual) for the standard **bcs**. It is a fairly simple procedure to define a set of macros to support such options. For example:

```
;       Alternate mnemonic macro
;       bge:    branch on greater than or equal to
;               (unsigned comparisons)
;
.macro          bge     branch_dest
                bcc     branch_dest
.endm
```

Operands

Many of these 6502 instructions support a variety of addressing modes, pushing the total number of operations (combinations of instructions and operands) up to 115. The following addressing modes are recognized by geoAssembler:

MODE	OPERAND FORMAT
implied	*(blank)*
relative	*addrexp*
accumulator	a
absolute	*zp-address*

absolute indexed X	*zp-address*,x; *addrexp*,x
absolute indexed Y	*zp-address*,y; *addrexp*,y
indexed indirect	(*zp-address*,x)
indirect indexed	(*zp-address*),y

For more information about 6502 instructions and operands, consult a book describing 6502 assembly language. Refer to Appendix D for a list of such books.

Comments

You can place a comment almost anywhere in your source code. It can share a line with other items such as labels and instructions, but it must always *follow* those items. A comment begins with a semicolon and extends to the end of a source line; geoAssembler ignores everything on the line after the semicolon. The only time a semicolon does not introduce a comment is when it appears within quotations, in which case it is considered ASCII string data.

```
;this line is a comment
lda   #55     ;this, too, is a comment
.byte "these; are; not; comments;"   ;but this is!
```

Expressions

Numeric Constants

geoAssembler will work with decimal (base 10), hexadecimal (base 16), octal (base 8), and binary (base 2) numbers in addition to character data. All numbers are considered to be 16-bit (two bytes) values for expression evaluation.

Decimal:	A string of decimal digits (0-9). Example: **1234**
Hexadecimal: digits	A dollar sign ($) followed by a string of hexadecimal (0-9, a-f). Example: **$4f9c**

Octal: A question mark (?) followed by a string of octal digits (0-7).
Example: **?07117**

Binary: A percent sign (%) followed by a string of binary digits (0,1).
Example: **%11001010**

Character: A single ASCII character enclosed in single-quotes (').
The character is converted to a 16-bit value with the high-byte set to zero.
Example: **'A'**

Notice that in an expression, the following would all be equivalent:

24930	(decimal)
$6162	(hex)
?60542	(octal)
%0110000101100010	(binary)
('a'*$100+'b')	(character)

Expression Evaluation

geoAssembler sports a full logical and arithmetic *expression evaluator* used to resolve operands and equates encountered in the source file. The expression evaluator is a standard algebraic parser which allows a wide variety of operators and nested parenthesization. It is much like the expression evaluator built into a standard C compiler or a BASIC interpreter.

An *expression* is any valid combination of symbols, numeric constants, and operators which geoAssembler can evaluate. geoLinker also supports this expression evaluator. This allows you to use complex expressions which contain external symbols (and, hence, cannot be evaluated at assembly-time) within your source code; geoLinker will evaluate them properly at link-time.

Arithmetic Operations

The expression evaluator uses 16-bit values for all its calculations. As an added benefit, it partially supports the two's-complement numbering system. Two's complement math allows positive and negative numbers but isn't true signed arithmetic; it's actually an artifact of binary math which allows addition and subtraction operations to "automatically" handle signed

and unsigned numbers because they are stored in the same 16-bit format. For example, a $fffe can represent 65534 or -2, depending on whether the number is considered to be signed or unsigned. For the majority of the cases, it won't matter whether you are dealing with signed values or unsigned values — The result will be correct. For example, the following two expressions will evaluate identically, even though one is signed arithmetic and the other is unsigned:

(-1) - (2)	*-1 equals $ffff* *2 equals $0002* *$ffff - $0002 = $fffd* *$fffd = -3*	**65535 - 2**	*65535 equals $ffff* *2 equals $0002* *$ffff - $0002 = $fffd* *$fffd = 65533*

However, there is a fly in the ointment. In cases of overflow (signed or unsigned) the result is truncated to 16-bits and no error is flagged. In short: if an unsigned value exceeds the range

0 <= *number* <= 65535

the value will truncate to 16 bits without flagging an unsigned overflow; if a signed value exceeds the range

-32768 <= *number* <= 32767

the value will truncate to 16 bits without flagging a signed overflow.

The expected result of adding $ffff to $ffff might be $1fffe if the arithmetic is considered unsigned, but this value cannot be contained in 16-bits, so the result is trunctated to $fffe, which just happens to be the correct signed result of (-1) + (-1), or -2, but an incorrect (truncated) unsigned result.

Also, when you are expecting a signed result and working with very large positive numbers or very small negative numbers, there is a possiblity that there will be a carry into the sign bit, resulting in what could be interpreted as a numeric overflow (example: -32768-5).

It is beyond the scope of this manual to document all the intricacies of two's-complement arithmetic. However, most assembly language books cover this topic in sufficient detail. Refer to Appendix D for reading recommendations.

Logical Operations

In addition to arithemtic operations, the expression evaluator can also

handle logical, or Boolean, operations. A logical expression is very much like an arithmetic expression, except that it has only two possible values: true or false. The result of a logical expression is called the *truth value* of the expression. Logical expressions are especially useful with conditional assembly directives such as **.if**.

Although logical and arithmetic operations are conceptually very different, the expression evaluator treats them similarly. The truth value of a logical expression is actually a numeric value. If the expression is true, it evaluates to an arithmetic one ($0001), and if the expression is false, it evaluates to an arithmetic zero ($0000). Conversely, if an arithmetic expression evaluates to non-zero, it is considered a logical true, and if it evaluates to zero, it is considered a logical false. This allows you to intermix logical and arithmetic operations within the same expression.

Operators

The following table shows all the valid operators and their precedence:

OPERATOR	PRECEDENCE	
( )	1	grouping parentheses (sub-expression)
-	2	unary negation
!	2	logical not
~	2	bitwise one's complement
[or <	2	low-byte
] or >	2	high-byte
**	3	exponentiation
*	4	multiplication
/	4	division
//	4	modulus
+	5	addition
-	5	subtraction
>>	6	logical shift right
<<	6	logical shift left
>	7	logical greater than
>=	7	logical greater than or equal to
<	7	logical less than
<=	7	logical less than or equal to
= = or =	8	logical equal
!=	8	logical not equal
&	9	bitwise and
^	10	bitwise exclusive-or (xor)
|	11	bitwise inclusive-or (ora)
&&	12	logical and
^^	13	logical exclusive-or
||	14	logical inclusive-or

(For information on typing-in certain operator symbols, refer to "Special Keystrokes" in Chapter 4.)

IMPORTANT: A common error is to use the BASIC logical not-equal operator (<>) instead of the geoAssembler !=. For example, if you used

.if (version<>c64)

instead of

.if (version != c64)

the expression evaluator would not recognize <> as a valid operator and would parse the expression as:

.if ((version) < (>c64))

or "if **version** is less than the high-byte of **c64**."

Evaluation

Expressions are evaluated based on operator precedence. Operators with lower precedence numbers are evaluated first, and operators with equal precedence are evaluated left to right. You can override operator precedence by grouping subexpressions within parentheses.

geoAssembler will ignore any whitespace between arguments and operators. Proper spacing can make complex expressions easier to read and understand.

Example expressions:

```
loop1
screen + $400 + %00001001
ram_start + buf_size-1
(mask1 | 1)<<4
((($8000<=offset1)&&(mask1<<12)) ||        ((]table&mask2)>40))
( ('p'-'A') + 2 )
```

Operator: ( )

Parentheses are used for grouping subexpressions in order to clarify or change the order of an expression's evaluation. For example, say we wanted to find which memory page (256-byte boundary) the address **bitmap** + **$3fff** evaluates to, we might try writing it as

```
bitmap + $3fff / 256
```

This would first divide $3fff by 256 (the number of bytes in a page) and add the result to **bitmap** because division (/) has a higher precedence than addition (+) — perfectly legal, but not what we wanted. We need to divide the entire expression by 256, not just the $3fff argument. We can use

parentheses to override the operator precedence.

```
( bitmap + $3fff )/256
```

Now $3fff is first added to **bitmap** and the result is then divided by 256 which provides us with the correct page number.

IMPORTANT: The standard round parentheses do double-duty in geoAssembler — they are used for both expression grouping and 6502 indirect addressing modes. This can pose a problem for the parser when it is unclear from context whether the parentheses are supposed to indicate grouping or indirection. For example, an ambiguous expression such as

```
lda   (label*5),y
```

could be interpreted as

```
lda   expression,y          ;absolute indexed
```

or as

```
lda   (expression),y        ;indirect indexed
```

In such cases, geoAssembler gives precedence to the addressing mode, which it establishes prior to sending the expression to the expression evaluator. If you do not want indirection, leave off the outer parentheses.

In order to speed assembly, geoAssembler establishes the addressing mode prior to parsing the expression. When looking for indirect addressing, geoAssembler does not actually go through and pair up matching parentheses (the job of the expression evaluator); rather, it merely looks for two outermost opening and closing parentheses in the operand. In most cases, these outermost parentheses do in fact indicate indirect addressing. However, this method is not foolproof — in some special cases, such as

```
lda   (addr+2)*(addr+3),y          ;indirect indexed
```

the parser sees the left- and rightmost parentheses and assumes indirect

addressing, even though, in the expression, these do not pair up. If you must include this type of expression in the operand, and you do not want indirect addressing, simply attach a monadic plus sign to the leftmost expression:

```
lda   +(addr+2)*(addr+3),y          ;absolute indexed
```

This way, the parser never encounters the leftmost parenthesis (it sees the + instead) and will therefore use absolute addressing, while the plus-sign has no effect on the eventual evaluation of the expression.

Operator: - (unary)

The unary minus sign simply negates the 16-bit sign of the number (two's complement negation). The expression **-10** is equivalent to **0-10**; it's as if the number were subtracted from zero.

Example:
-16 is equivalent to $fff0

Operator: ~ (unary)

This unary operator yields the bitwise one's complement of a number by reversing all 16 bits. All 1 bits become 0 and all 0 bits become 1. It is equivalent to exclusive-or'ing a value with $ffff (-1).

Example:

```
~%0000111101010011      ($0f53)
      yields
%1111000010101100      ($f0ac)
```

Operators:], [, <, > (unary)

These operators extract the high- or low-byte from a two-byte number.] and > extract the high-byte; [and < extract the low-byte.

Examples:

```
]$fe34      yields   $fe
[$fe34      yields   $34
>$783e      yields   $78
<$783e      yields   $3e
```

These operators are especially useful for dealing with two-byte addresses as in:

```
;store address of ISR routine into a jump vector
sei                 ;stop all interrupts
lda   #[isr         ;get low byte
sta   isr_vec       ;set into vector in low/high order
lda   #]isr         ;get high byte
sta   isr_vec+1
cli                 ;reenable interrupts
```

Operator: **

The exponentiation operator allows you to raise a number to an integer power. The exponentiation, as with other operations, is restricted to the range of a 16-bit signed integer.

Example:
28** is equivalent to raising two to the eighth power (2^8 = $100).

Operator: //

The modulus operator provides the remainder of integer division. For example, 21 modulo 5 results in the remainder of 21 divided by 5; since 5 divides into 21 four times with a remainder of one, 21 modulo 5 is 1.

Example:
35//$a is equivalent to the remainder of 35 divided by 11, or 2.

Operators: *, /, +, -

These standard arithmetic operators (multiplication, division, addition, and subtraction) all operate on 16-bit numbers. Addition and subtraction will take advantage of the two's complement numbering system, allowing positive and negative numbers and will, therefore, not generate overflow errors. Multiplication and division are unsigned. Multiplication overflow will generate an error. The division operator is purely integral, thereby discarding any remainder or fractional portion of the result.

Operators: >>, <<

These operators shift the argument on the left of the operator the number of times determined by the argument on the right of the operator. << is a left shift and >> is a right shift. The shifts are not arithmetic, so there is no sign-extension. Bits shifted out of the 16-bit integer are lost. Zeros are shifted in.

Examples:

```
%0001<<3        shifts %0001 left 3 times, resulting in %1000
$ffce>>4        shifts $ffce right 4 times, resulting in $0ffc

(high_byte<<8) & (low_byte)
```

Operators: &, |, ^

These bit operators perform and, or, and exclusive-or operations (respectively) on the binary values of two arguments. They are analagous to the 6502 **and**, **ora**, and **eor** instructions. & (and) yields a one-bit in the result wherever there is a one-bit in both arguments; | (or) yields a one-bit in the result wherever there is a one-bit in either arguemnt; ^ (exclusive-or) yields a one-bit in the result wherever there is a one-bit in either argument but not in both.

Examples:

```
%1100 & %1010     yields %1000
%1100 | %1010     yields %1110
%1100 ^ %1010     yields %0110

(digit1 & $000f) | (digit2<<4 & $00f0)
```

Operator: !

Pronounced "not," this unary logical operator negates the truth-value of an expression. If the expression is true (non-zero) it evaluates to false (zero); if the expression is false (zero) it evaluates to true (one).

Examples:

```
False       = 0             ;equate to false
True        = !False        ;set to opposite truth value
.if    !debug_mode          ;if not in debug mode...
```

Operators: >, >=, <, <=, ==, =, !=

These standard comparison operators compare two 16-bit unsigned integer expressions and evaluate to either logical true (one) or logical false (zero). They are most often used in conditional assembly, but can appear in the context of any expression. The single and double equal sign are interchangeable as comparison operators

Examples:

10 > 6	evaluates to true (10 greater than 6)
%110 >= $22	evaluates to false (greater than, equal to)
$fe < 100	evaluates to false (less than)
?60542 <= $6162	evaluates to true (less than, equal to)
$fc == 252	evaluates to true (equal to)
$ffff = -4	evaluates to false (equal to)
12 != 12	evaluates to false (not equal to)

```
.if  (disk_buf > (10 * $400))      ;if greater than 10K...
```

NOTE: The > and < logical symbols operate with pairs of expressions; they act quite differently in a unary context (high- and low-byte operators).

Operators: &&, ||, ^^

These logical operators perform and, or, and exclusive-or operations (respectively) on the truth-value of two expressions. && (and) evaluates true if both expressions are true; || (or) evaluates true if either expression is true; ^^ (exclusive-or) evaluates true if one expression is true and the other is false.

Examples:

```
.if      (buffer_size >= 100) && (buffer_size <1000)
```

If the buffer size is greater than or equal to 100 and it's also less than one thousand, then...

```
.if      (data > 1000) || (buf_flag) || (free_space < (20 *          $400))
```

If the data size is greater than 100 or the buffer flag is set to true or there is less than 20 Kilobytes of free space, then...

.if (debug ^^ test)

If the debug flag is set or the test flag is set (but not both), then...

Mixing Logical and Arithmetic Expressions

Logical and arithmetic expressions may be intermixed. Logical expressions evaluate to either an arithmetic one (1) if the expression is true, or an arithmetic zero (0) if the expression is false. Conversely, if an arithmetic expression evaluates to non-zero, it is considered a logical true, and if it evaluates to zero, it is considered a logical false. Arithmetic and logical operators can even be used within the same expression. As an example, consider the following:

```
buf_space =  drives*(cache_siz+((disk>K_thresh)*big_buf))
```

Notice the logical subexpression **(disk>K_thresh)** buried within the expression. If **disk** is greater than **K_thresh**, then the subexpression will evaluate to true, and its arithmetic value of one will be used as a multiplicand to include the value of **big_buf**. However, if **disk** is less than or equal to **K_thresh**, then the subexpression will evaluate to false, yielding an arithmetic value of zero, and preventing the value **big_buf** from being added into the expression.

NOTE: relying on the arithmetic value of a logical expression (as in the above example) is sometimes considered bad programming practice. The same result can always be realized with multiple expressions and conditional assembly.

Directives

Directives, often called pseudo-ops, instruct geoAssembler to perform some action, such as include another source file, begin a macro definition, or define an equate. Other than **.byte** and **.word**, directives do not generate any object code. Most directives are preceded by a period (.) to distinguish them from macro names and 6502 mnemonics.

Summary of Directives

The following directives are recognized by geoAssembler.

Assembly Control

.include	Include another source code file into the assembly.
.zsect	Begin zero-page section.
.ramsect	Begin unitialized data section.
.psect	Begin program section (default).
.echo	Echo text to the error file.
.end	End assembly (optional at end of source file).

Symbols

=	define equate; do not send symbol to debugger.
= =	define equate; send symbol to debugger.
.eqin	Begin sending equates to the linker (default).
.noeqin	Stop sending equates to the linker.
.glbl	Begin sending global labels to the linker (default).
.noglbl	Stop sending global labels to the linker.

Data

.byte	Include byte-sized data and strings into psect section.
.word	Include word-sized data into psect section.
.block	Reserve space.

Conditional Assembly

.if	Start conditional; assemble if expression is true.
.else	Assemble if expression was false.
.elif	Start new conditional if expression was false.
.endif	End conditional.

Macro Definition

.macro	Begin macro definition.
.endm	End macro definition.

Header Definition

.header	Begin header definition.
.endh	End header definition.

Assembly Control Directives

Directive: **.include**

Purpose: Includes source code from another file directly in-line with the current assembly.

Usage: **.include** ***filename***

Note: The *filename* must be a valid geoWrite source file. If you have two drives (one can be a RAMdisk), geoAssembler will automatically search both for the desired file, starting with the same disk as the current assembly file.

When geoAssembler encounters a **.include** directive, it suspends assembly of the current file and begins reading source lines from the specified include file just as if they were part of the original assembly file. When geoAssembler encounters the end of the include file or a **.end** directive, it returns to the previous assembly level and continues with the line following the **.include.**

Include files may themselves have **.include** directives. However this file *nesting* may only extend to a depth of three. That is: you may only have three levels of files (counting the main assembly file) which include other files. Any including beyond this limit will generate an error.

Example:

```
.include    macros
.include    zpage
.include    equates
.include    maincode
.include    subroutines
(Note: this is not an example of nesting)
```

Directive: **.zsect**

Purpose: Begins zero-page definitions section.

Usage: **.zsect** [*zp-address*]

Note: *zp-address* is an optional zero-page absolute address ($00-$ff); If the address is omitted, geoAssembler will use the current value of the zsect location counter. At the start of an assembly, the zsect location counter is initialized to $00.

A zsect section is essentially a zero-page version of ramsect section; geoAssembler maintains a separate section for zero-page variables because zero-page references must be resolved during the first pass of the assembler. For this reason, the zsect section, unlike the ramsect section, cannot be relocated and must be given an absolute address at assembly-time; there is no **.zsect** linker command.

Because of the way zero-page references are handled, they must be defined before they are actually used; they are evaluated during the first pass of the assembler and cannot be left to the linker for resolution, nor can they be forward-referenced. This poses a problem for multiple source files which access the same zero-page variables because you cannot rely on linker resolution as you can with non-zero-page addresses. The best way to handle this is to **.include** a zero-page definition file into the assembly of each source module, treating zero-page variables as if they were equates.

.zsect begins a zsect section and it extends until the next **.ramsect** or **.psect** directive. Source code in a zsect section cannot generate any object code. This means that 6502 opcodes, **.byte**, and **.word** will all generate errors within a zsect section.

.zsect is used in combination with the **.block** directive, allowing the zsect location counter to be incremented and variable space to be reserved.

NOTE: It is not necessary to include zero-page equates (as opposed to labels) within a zsect section. geoAssembler is smart enough to use zero-page addressing when an equated constant is less than $100 is used as an address.

Example:

```
        .zsect      $70     ;begin zp variable space
;       ZERO PAGE VARIABLES
counter:    .block 2
pointer:    .block 2
temp1:      .block 1
temp2:      .block 1
temp3:      .block 1
X_coords:   .block 4
Y_coords:   .block 4
        .psect              ;end zsect section and begin psect
```

Directive: **.ramsect**

Purpose: Begins unitialized data (non-zero-page) section.

Usage: **.ramsect** [*addrexp*]

Note: *addrexp* is an optional absolute address within the 6502's addressing space ($0000-$ffff). If the address is an expression, it must evaluate on the first pass of the assembler — the expression may not contain any external symbols, nor any relocatable, external, or unresolved labels. If an address *is not* specified, it will be left to the linker to relocate the data area; If an address *is* specified, the current and all subsequent ramsect definitions will be assigned absolute addresses at assembly-time — previous ramsects (without addresses) will be unaffected and will still be relocated by the linker.

A **.ramsect** directive begins a ramsect section, which extends until the next **.psect** or **.zsect** directive. Source code in a ramsect section cannot generate any object code. This means that 6502 opcodes, **.byte**, and **.word** will all generate errors within a ramsect section.

All labels within a ramsect section are assigned the current value of the ramsect counter. In most cases, you will want the absolute value of these data areas to be determined by the linker, which it will do automatically if no absolute address is specified. However, sometimes it is desirable to assign an absolute value to the ramsect counter during assembly. In these cases, simply follow the **.ramsect** with a valid absolute address — all subsequent labels in **.ramsect** sections will be assigned values based on this address. In either case, ramsect sections take up no space in the eventual application file; they are merely placeholders during the assembly-link process.

Like the zsect section, the **.block** directive is used to increment the object code counter and reserve data space.

IMPORTANT: When your program is executed, the values in ramsect data areas are unknown and should not be used without first initializing them. An ideal way to initialize **.ramsect** variables is with the GEOS **InitRam** routine.

Example:

```
        .ramsect    ;begin variable/data space -- let linker
resolve
    ;     VARIABLES
    timer:      .block 3
    X_pos:      .block 2
    Y_pos:      .block 2
    Coldstart:  .block 1
    ;     DATA BUFFERS
    disk_buf:   .block $100
    scrn_buf:   .block $1000
    scratch:    .block 16
        .ramsect scrn_RAM           ;start new ramsect
(absolute)
    Foreground:     .block $1000
    Background:     .block $1000
        .psect      ;end of data space, start of program area
```

Directive: **.psect**

Purpose: Begins program code and initialized data section.

Usage: **.psect**

Note: Unlike **.zsect** and **.ramsect**, **.psect** will not accept an absolute address. Psect sections are *always* relocated to an absolute address by the linker.

When geoAssembler starts processing a file, it defaults to the psect section. The psect section contains all opcodes and initialized data — essentially anything which will generate object code (6502 source code, **.byte**, **.word**, etc.).

When geoAssembler begins, the psect location counter is set to zero. As it passes through the source code, it increments this counter to accomodate the object code generated. All labels within the psect section are assigned the current value of the psect counter. At link-time, these *relocatable* values are changed to absolute values in the relocation process.

NOTE: The **.block** directive can be used within a psect section; it will generate a block of zeros ($00) in the object code.

Example:

```
        .psect          ;start program section
Initbufr:   ldx   #$00
1$:         lda   initdata,x      ;initialize the data buffer
            sta   buffer,x
            inx
            cpx   #idata_size     ;only copy proper # of     bytes
            bne   1$
            rts
;
initdata:   .byte     $34, $54, $10, $f5, $ff, $a0, $a3
            .byte     $90, $0d, $f1
idata_end:      ;placeholder for end of data
idata_size  =   idata_end - initdata
;
        .ramsect    ;start ramsect section for buffer space
buffer:     .block idata_size
;
        .psect      ;end ramsect and return to psect
```

Directive: **.echo**

Purpose: Sends user-defined text to the error file.

Usage: **.echo** ***text***

Note: *text* is up to a full line of ASCII text. No quotes are required.

The **.echo** directive sends a line of text to the .err file generated by geoAssembler.
This allows you to generate your own messages, warnings, and errors which will be written to the error file. This won't actually create assembly errors, however — the error count doesn't actually change — only the text is sent to the file.

Example:

```
.if     debug
        .include dbgcode
.else
        .echo   Warning: debugging code not installed
.endif
```

Directive: **.end**

Purpose: Ends the current level of assembly — if in an include file, geoAssembler resumes processing of the parent file; if in a main assembly file, geoAssembler ends the assembly.

Usage: **.end**

The **.end** directive is entirely optional because the normal end-of-file marker in geoWrite files will alert geoAssembler to end the current level of assembly. It is included here mainly for historical purposes.

Symbol Directives

Directive: =, ==

Purpose: To equate a value to a symbol.

Usage: ***symbol*** = ***exp***

symbol == ***exp***

Note: *symbol* is a valid symbol name followed by a colon (:) and *exp* is an expression which evaluates to an absolute value at assembly time. An equate may be either an address or a constant.

The = and == directives assign absolute, constant values to symbols which may later be used within expressions. Equates make your source code easier to read and understand, as well as maintain. They allow you to use descriptive names for constant values (e.g., **NULL** for $00 or **FF** for an ASCII form-feed) and addresses. Additionally, if you use the equate consistently, you need only change the symbol definition to affect a change throughout the entire program.

The == directive will cause the symbol to be included in the symbol table used by the debugger; the = will cause the symbol to be excluded from the symbol table used by the debugger.

IMPORTANT: Equates must be resolvable on the first pass of the assembly. This means you cannot use any forward or external references in an equate's definition; the expression cannot contain any symbols which have not yet been defined, regardless of whether they are defined later in the current file or during the link-stage.

NOTE: Whether or not equated symbols are sent to the linker can be controlled with the **.eqin** and **.noeqin** directives. If the **.noeqin** option is in effect, even symbols equated with the == directive will not make it beyond the assembly-stage.

Directive: **.eqin, .noeqin**

Purpose: To allow or suppress equate passing to the linker.

Usage: **.eqin**

.noeqin

Note: No parameters.

geoAssembler, by default, passes all equates to the linker. At times it is desirable to prevent this from happening to certain symbols, to limit the scope of these equates to the current assembly file.

.noeqin instructs geoAssembler to stop sending equates to the linker. All subsequent equates, up to a following **.eqin** directive, will not be sent to the linker. They can only be accessed from within the current assembly file. They will be invisible to any other .rel files which are later linked.

.eqin instructs geoAssembler to once again send equates to the linker.

NOTE: Because equates suppressed with the **.noeqin** directive will not be sent to the linker, they will also never get sent to the debugger regardless of whether the = or the == directive is used.

Example:

```
; --- sent to linker and debugger ---
sector:     ==    $01
track :     ==    $5c
buf_addr:   ==    $3000
;
; --- sent to linker but not debugger ---
EOF:        =     -1
EOL:        =     $4c
;
; --- not sent to linker nor to debugger ---
.noeqin
start_cnt:  =     $ff
retries:    =     $0a
home:       ==    track*2
.eqin
```

Directive: **.glbl, .noglbl**

Purpose: To allow or suppress global labels passing to the linker.

Usage: **.glbl**

.noglbl

Note: No parameters

By default, geoAssembler passes all labels to the linker. At times it is desirable to prevent this from happening to certain symbols, to limit the scope of these labels to the current assembly file.

.noglbl instructs geoAssembler to stop sending labels to the linker. All subsequent labels, up to a following **.glbl** directive, will not be sent to the linker.

.glbl instructs geoAssembler to once again send labels to the linker. They can only be accessed from within the current assembly file. They will be invisible to any other .rel files which are later linked.

NOTE: Because labels suppressed with the **.noglbl** directive will not be sent to the linker, they will also never get passed to the debugger.

Example:

```
;--- send these labels to linker ---
.glbl
Start:
        .include maincode
;
jump_tbl:
        .word       Draw_box, Move_icon, Call_extern
        .word       Copy_buf, Read_mouse, Pterm
;
;--- suppress sending these to linker ---
.noglbl
local_jumps:
        .word       box_remove, mouse_reset, abort
warmstart:
        .include main2
.glbl
```

Data Directives

Directive: **.byte**

Purpose: Deposits byte-sized data directly into the object code.

Usage: **.byte** ***exp|string{,exp|string}***

Note: *exp|string* refers to either a valid expression or an ASCII string enclosed in double-quotes.

The **.byte** directive inserts data bytes directly into the object code and increments the psect counter appropriately. **.byte** can only be used within a psect section. With expressions that exceed the capacity of one byte (>$ff), only the low-byte of the value will be used, and a warning will be generated. To explicity extract the low-byte, use the < or [operator; to extract the high-byte, use the] or > operator.

String data enclosed in double-quotes will generate the ASCII equivalent for each character in the string, one byte per character.

Examples

```
string1:    .byte "This is a sample string", CR, LF,        NULL
data1:      .byte $ff, %0101111, "hello", $56, $34+'@'
Hi_jmp:     .byte ]addr1, ]addr2, ]addr3, ]addr4
Lo_jmp:     .byte [addr1, [addr2, [addr3, [addr4
```

Directive: **.word**

Purpose: Deposits word-sized data (two bytes) directly into the object code in 6502 low-byte, high-byte order.

Usage: **.word** ***exp*{,*exp*}**

Note: *exp* refers to a valid expression. Strings are not used.

The **.word** directive inserts data words directly into the object code and increments the psect counter appropriately. A word is two consecutive bytes, and, on the 6502, the low-byte is stored first. **.word** is usually used to store address data for jump tables.

Note that

```
.word       $12fe                ;low followed by high
```

is equivalent to

```
.byte       [$12fe,]$12fe        ;low followed by high
```

Byte-sized data stored with the **.word** directive will have the high-byte set to $00.

Examples:

```
.word       jump1, jump2, jump3, jump4, jump5, $00
.word       addr1, addr2, addr3, ($5000+addr4)/2+1
```

Directive: **.block**

Purpose: Reserves unitialized data space in zsect and ramsect sections. Can also be used to generate blocks $00 bytes in a psect section.

Usage: **.block** ***exp***

Note: *exp* refers to a valid expression which determines the number of bytes to actually reserve. The expression must be resolvable when it is encountered on the first pass and cannot contain external or relocatable symbols.

.block is used within zsect and ramsect sections to reserve byte-sized space without actually generating any object code data. It merely increments the appropriate zsect or ramsect counter by the specified number of bytes.

NOTE: **.block** will generate a block of zeros ($00) when used within a **.psect** section.

Example:

```
        .zsect $30
critic:     .block 1
on_flag:    .block 1
timer_3:    .block TIMER_size*2
date:       .block 4
        .ramsect
temp1:      .block 2
temp2:      .block 2
U_right:    .block 2
L_left:     .block 2
buffer:     .block 3000+(disk_K*$100)
buf_flag:   .block 1
screen:     .block $8000
        .psect
```

Conditional Assembly

Conditional assembly allows you to have specific sections of source code automatically included in or removed from the assembly based on the truth-value of an expression. This allows you to use the same source code files to assemble different versions of the same application. For example, during program development, you might build diagnostic code into the application, code which will display the program's status and other debugging information. This code is unnecessary in the final version, though. One elegant way of handling this is to surround your debugging code with conditionals. During development, you set an equate in the main assembly file which causes these conditionals to evaluate to true, thereby including the diagnostic routines. In the final version, you need merely change the value of the equate so that the conditionals don't succeed and the code is not assembled.

Directive: **.if, .else, .elif, .endif**

Purpose: Conditional assembly directives; Instruct geoAssember to either include or ignore specific lines of assembly code based on the truth-value of an expression.

Usage: **.if** ***exp***
[.else|.elif ***exp*****]**
.endif

Note: *exp* is a valid expression, usually a logical expression.

The **.if** directive begins a conditional section. If the expression evaluates to false, assembly is suppressed until geoAssembler encounters an **.else**, **.elif**, or **.endif**. At that time, assembly is resumed or not depending on the directive encountered. If the expression is true, geoAssembler continues assembling. You can think of a conditional like this: "If the expression is true, then the following source lines will be assembled..."

geoAssembler determines the truth-value of the expression using the standard logical expression evaluator. A zero value is considered to be false; a non-zero value is considered to be true.

The simplest use of a conditional consists of an **.if** followed by some source lines which end with an **.endif**. If the **.if** expression is false, the code between the two directives will be left out of the assembly; if it is true, they will be included.

Example:

```
.if ( buffer>=$3000 )          ;conditional
;*** this code is only assembled if buffer>=$3000
        lda     #M_on
        sta     semaphore
        jsr     xtra_buf
        jsr     Malloc
.endif                         ;end of conditional
;*** assembly is now back to normal...
```

The **.else** directive allows you to set up two mutually-exclusive sections of code, one (and only one) of which will be included in the assembly. Think of the **.else** directive as: "*If* the expression is true, assemble this chunk of code... *else*, it must be false, so assemble this..."

Example:

```
    .if (diagnosis == ON)       ;if the diagnosis code is
desired...
        .include WhatsUp        ;then...
        lda     #flag_ON        ;    assemble
        sta     fallout         ;            this
        sta     crash           ;                stuff...
        jsr     breakpts
    .else                       ;otherwise, use this code
instead...
        lda     #flag_OFF       ;-- this gets assembled only if
        sta     fallout         ;       (diagnosis != ON)
        sta     crash
    .endif                      ;end of conditional
```

The **.elif** directive is merely a combination of the **.else** and the **.if** conditionals. It allows an **.else** to trigger another conditional. In this case, the additional **.if** implicit in the **.elif** requires its own corresponding **.endif** and may, itself, use additional **.elif**'s.

```
.if debug                       ;if using debugging code...
        lda     #flag_ON        ;then...
        sta     dbug_flag
        .if (dbug_level == 1)           ;then, if level 1...
        .include dlevel1
        .elif (dbug_level > 10)         ;else if level >10...
                .include breakcode
                .else           ;else tied to the if in elseif
                lda   #flag_OFF
                sta   brk_flag
                .endif          ;end of elseif
        .endif              ;end inner if
.endif                  ;end outermost if
```

This tortuous example shows some of the complexity you can achieve by nesting conditionals. Note, however, conditionals can only be nested to a level of ten deep. Sometimes it helps document what you're doing if you indent the levels of nesting (as above) to illustrate the hierarchy.

Macros

Be forewarned: macro programming is an advanced topic, especially for somebody new to 6502 assembly language. If macros seem confusing, don't worry. Master assembly language first, then come back and study macros. They can save time and make your source code more maintainable and compact.

What is a Macro?

At their simplest level, macros are merely an advanced form of text substitution, and they are purely a function of the assembler. If you have a common or complex chunk of code, you can assign it a name or abbreviation. This is called defining the macro or *macro definition.* Now, each time you want to use this code, rather than type in the actual source lines, you simply use this abbreviation. geoAssembler will recognize the abbreviation as a macro *use*, or *invocation*, and will replace it with the previously defined source code, thereby *expanding* the macro name to its full definition. Once you have defined a set of useful, general purpose macros (as we have in the sample macro file), you may include them as library files in all your assemblies.

And What's This About Parameters?

One of the features that makes macros so powerful is that you can pass parameters to them. That is, when you invoke the macro, you can pass it label names, variables, constants, flags, addressing modes, and the like; geoAssembler will take these parameters and insert them into the actual macro-expanded code as determined in the macro definition. You might call a macro like this:

```
SuBW    subtrahend, minuend    ;subtract word
```

At assembly-time geoAssembler will expand the macro (defined earlier in the source code) to produce something like this:

```
lda   minuend       ;get byte value
sec
sbc   subtrahend    ;subtract low byte
sta   minuend       ;overwrite minuend with result
lda   minuend+1     ;high-byte with carry
sbc   subtrahend+1
sta   minuend+1
```

All this is done automatically! You will not actually see this expansion. However, this is how it will look to the assembler.

Directive: **.macro, .endm**

Purpose: For defining macros.

Usage: **.macro** ***name*** **[*parameter*{,*parameter*}]**
macro definition
.endm

Note: *name* is the macro name — you will use this for all invocations of the macro — it can be any valid symbol; *parameter* is an optional parameter declaration. If you expect parameters, you must delcare them.

Important: The following directives are invalid within a macro definition: **.macro**, **.endm**, **.include**, or **.end**. They will generate errors.

The **.macro** directive tells geoAssembler that all code up to the next **.endm** (end macro) directive is part of the macro definition. The **.macro** is followed by the name of the macro (the abbreviation which you later use to invoke it). After the macro name you may declare from zero to six parameters, separated by commas.

The body of the macro consists of normal geoAssembler source code. You may use mnemonics, most directives, even previously defined macros, within the macro definition. You can use the parameter names anywhere in the this source code — wherever you would like the parameter name replaced with the actual parameter passed to the macro upon invocation. geoAssembler does absolutely no syntax checking prior to parameter substitution, so there is little you are unable to pass it: strings, labels, characters, equates, and expressions are all fair game.

Follow the body of the macro with an **.endm** directive to indicate the end of the macro definition.

Once a macro has been defined, it may be invoked in your source code by placing the macro name in the opcode field of the source line and any parameters in the operand field, separated by commas. geoAssembler recognizes the macro name as a macro invocation and expands it appropriately.

First, geoAssembler takes any parameters in the invocation and inserts them in the appropriate places in the macro body. The macro body, with the parameters in their proper places, is fed directly into the assembler's input stream exactly as if the macro body was part of the source code. geoAssembler will then attempt to assemble on a line-by-line basis, flagging errors as normal. When the end of the macro body is reached, geoAssembler again resumes assembling with the next line in the source code. In this way, much like an **.include**, one macro line can be expanded to almost any number of actual source lines. Keep this in mind when using large macros — if you use them often enough, they may warrant an actual subroutine to save memory space.

As an example, we will define and then invoke a macro from the sample macro file. The following is the macro definition for the **AddVW** macro. It adds a one or two byte constant value (immediate value) to a word (two bytes) in memory. The word in memory is stored in 6502 low, high order. The macro uses conditional assembly to handle one and two byte constants differently, generating the most efficient code for each case.

```
;***********************************************
;
;       Add Value to Word:      AddVW    value, dest
;
;       Args:   value:      constant to add to dest
;               dest:       address of word to add to
;
;       Action: dest = dest + value
;
;***********************************************
.macro  AddVW     value,dest
        clc                     ;must add with carry
        lda     #[(value)       ;add low byte first
        adc     dest+0
        sta     dest+0          ;and replace with low of result

;If the value to add is only one byte, then special-case the carry
.if     (value >= 0) && (value <= 255)
        bcc     noInc           ;if low-byte generated a carry...
        inc     dest+1          ;   then, increment high-byte
noInc:
.else   ;value larger than one byte, so do full word add
        lda     #](value)       ;get high byte
        adc     dest+1          ;add high byte in with carry
        sta     dest+1          ;and replace with high of result
.endif
.endm
```

After we have defined this macro, we can then invoke it from within our source code:

```
        AddVW   $20, Sum1               ;add $20 to Sum1
        AddVW   3000, Sum2              ;add 3000 to Sum2
```

During assembly, when geoAssembler encounters these invocations, the macro will be expanded and the parameters will be substituted. geoAssembler would expand the first usage (**AddVW $20, Sum1**) like this:

```
        clc
        lda     #[($20)
        adc     Sum1+0
        sta     Sum1+0
        bcc     9999$
        inc     Sum1+1
9999$:
```

First, notice that the constant (**$20**) and the variable (**Sum1**) were substituted into the macro definition for **value** and **dest**, respectively. Also notice that because the constant (**$20**) was a one-byte expression, the conditional in the macro evaluated to true and generated the code between the **.if** and the **.else**. Finally, notice the macro label **noInc** was replaced with the local label **9999$**; this will be explained later.

The second invocation would be expanded like this:

```
        clc
        lda     #[(3000)
        adc     Sum2+0
        sta     Sum2+0
        lda     #](3000)
        adc     Sum2+1
        sta     Sum2+1
```

In this case, because the constant (**3000**) was a two-byte value, the conditional evaluated to false, and the code between the **.else** and the **.endif** was included instead of the code between the **.if** and the **.else**.

Macro Names

Each macro name must be unique and it must conform to the geoAssembler symbol notation. A macro name may be up to 20 characters long, of which only the first eight are significant. It must begin with an alpha character, but the remaining characters can consist of numbers and the underscore (_) symbol. Case is significant. Although it is not a good idea, you can have a label and a macro of the same name; geoAssembler can distinguish the two from context.

Parameters and Parameter Names

A macro can accept from zero to six parameters which must be declared in the macro definition. Parameter names can be up to ten characters long, all

of which are significant. As in symbols and macro names, case is also significant. Parameter names may begin with the underscore symbol (_), which allows you to prevent accidental conflicts with labels, equates, or macros within the macro definition. If two names do coincide, macro substitution will take precedence.

Parameter Substitution

When a macro is invoked, the parameters passed to the macro (unique for each invocation) are substituted into the body of the macro according to the parameter names (which are in the definition). In the macro invocation, parameters follow the macro name and are separated by commas. geoAssembler does a straight text substitution, so internal spaces and other characters are maintained throughout the substitution. This lets you pass entire expressions like

```
(value * 35 + (%1010<<2)) + $33
```

or even entire opcodes and operands like:

```
adc   #$2e
```

The only complication occurs when you need to pass a parameter which contains a comma, such as an indexed addressing operand:

```
addr,y
```

geoAssembler will interpret this as two separate parameters: **addr** as the first paramter and **y** as the second. You can get around this problem by enclosing the entire parameter in double-quotes:

```
"addr,y"
```

geoAssembler will strip the quotes and substitute the entire string. Notice that this also allows you to send string data (for **.byte** statements) by enclosing the string in two sets of quotes:

```
""this string will be substituted""
```

The outside set of quotes will be stripped in the macro invocation, but the inside set will be substituted along with the rest of the string. This allows you to do something like

```
.byte       parameter
```

in a macro and pass it either a string or a value in the invocation.

Parameter substitution will occur anywhere geoAssembler finds the parameter name in the macro body, except when the name is within quotes, in which case geoAssembler assumes it is part of a string, or in the opcode field of a source line.

Too Few or Too Many Parameters

When a macro is invoked, any extra parameters will be ignored in the expansion. That is: if you pass more parameters than were declared in the macro definition, the extra parameters will be discarded. If you pass less parameters than were declared in the macro definition, the undefined parameters will be set to a logical false (zero). This allows you to send a variable number of parameters and generate the appropriate code with conditional assembly.

Labels Within Macros

geoAssembler has a unique way of handling labels within a macro body. When the macro is expanded, geoAssembler tries to convert any labels within the macro to local labels. There is a macro local label counter which begins at 9999$ and decrements for each label that is used within a macro (at each invocation). That label, and all uses of that label within the macro, are replaced with the value of this counter, thereby converting them to local labels. These labels will be treated exactly like normal local labels when you invoke the macro within your source code. Each label in each macro expansion will have a unique local label value, so there won't be any conflicts between macros. However, there is one potential source of conflict: if you use large-value local labels in your normal source code (like 9984$), it might conflict with a nearby macro expansion, thereby producing a duplicate local label error. To prevent this from happening, avoid using large-value local labels — if you stay away from four-digit numbers beginning with "9", there should not be any problems.

There is one special-case where a label in a macro is not automatically transformed into a local label: when the label is actually a parameter slated for substitution. If, for example, you have a macro like:

```
.macro    Double_lp   label_1, label_2, yvalue, xvalue
          ldy    #yvalue
```

```
label_2:
        ldx     #xvalue
label_1:
.endm
```

where the label **label_1** and **label_2** are actually parameters, **label_1** and **label_2** will not be converted to local labels. Instead, the macro will expect valid symbols or local labels to be passed to it as the first two parameters. For example

```
Double_lp   inner_loop, outer_loop, $35, $ff
```

would expand as

```
        ldy     #$35
outer_loop:
        ldx     #$ff
inner_loop:
```

and this would allow you to use the label name globally later in the program as in

```
Double_lp   inner_loop, outer_loop, $4e, $54
sta     table,x
dex
bne     inner_loop
inc     table+1
dey
bne     outer_loop
```

Note that you just as easily could have passed local labels instead of global labels as in

```
Double_lp   10$, 11$, $54, $ff
```

where the first two parameters (**10$** and **11$**) are local labels. The macro

would expand to:

```
        ldy     #$54
11$:
        ldx     #$ff
10$:
```

Immediate Mode and Constant Values

The expression evaluator will ignore any # signs within expressions. This allows immediate mode addressing and constant parameters to be handled flexibly within a macro expansion. For example, with the following macro

```
;************************************************
;
;       Add Value to Byte:      AddVB     value, dest
;
;       Args:   value:      byte constant to add to dest
;               dest:       address of byte to add to
;
;       Action: dest = dest + value
;
;************************************************
.macro  AddVB       value, dest
        lda     dest
        clc
        adc     #value
        sta     dest
.endm
```

We can invoke this macro with either of the following:

```
        AddVB       #$ff, total
        AddVB       $ff, total
```

In the first case, the parameter substitution will generate an extra # sign,

resulting in the line

```
adc     ##$ff
```

The expression evaluator will drop the unneeded # sign. In the second case, there will only be one # sign, so the interpretation is trivial. This way, if we call **AddVB** with a constant value like

```
AddVB   #constant, total
```

It will be clear that the constant will be used in an immediate-mode context.

Macro Nesting

Macros can invoke other macros. In fact, they can even (recursively) invoke themselves. However, this macro nesting is limited to three levels. You cannot *define* a macro inside another macro.

Macro Overflows

geoAssembler maintains a number of tables for macros, all of which are of limited size, but large enough to handle the majority of cases. You will probably never encounter a macro overflow error unless you are nesting groups of macros with a large number of parameters and internal labels. For more information on macro errors, refer to Appendix E.

Header Definition

Directive: **.header, .endh**

Purpose: These special directives allow you to create a GEOS file header data structure for your GEOS application.

Usage:
```
.header
.word       $00       ;last block -- always 0
.byte       3         ;icon width -- always 3
.byte       21 ;      icon height -- always 21
;icon information follows

                      Seq

.byte       C64type   ;usually $83
.byte       GEOtype   ;GEOS file type
.byte       GEOstruct ;GEOS structure type
.word       FileStart ;load address
.word       FileEnd   ;end of app. address
.word       InitProg  ;init. address
.byte       filename  ;must be 20 bytes
[.byte      ...]      ;optional fields
.endh
```

The **.header** and **.endh** directives invoke an additional level of error-checking for creating a GEOS file header. The header involves a very rigid syntax and critical byte counts which are checked automatically by geoAssembler.

The header directives don't actually create a header on the disk. Rather, they build a 256-byte data structure into a normal .rel file. This structure can then be used by geoLinker to create the header for your application file. In this case, the header should be the only item in the source file. All other data will be ignored by the linker.

HINT: when you are first building an application, use the default header by omitting the **.header** directive from the link command file; in the final stages of development, you can then build your customized header.

The header directives can also be used to create prototype headers for use inside your applications. Simply include the header directives in a **.psect** section where you would like data to be generated. A 256-byte block will be created.

The area between the **.header** and **.endh** follows a strict syntax. 6502 mnemonics, **.psect**, **.ramsect**, **.zsect**, **.include**, **.macro**, **.endm**, **.end**, or **.header** are invalid within the header definition. And any data-creation directives (**.word**, **.byte**) must be in the order and format as described.

Header Syntax

The syntax checker for header definitions is primarily a byte counter. It has a table of **.byte** and **.word** definitions which it checks against and will generate an error if it doesn't find what it expects. You may, however, include most types of directives, lablels, and equate definitions, even macro invocations, within the source code, as long as the actual data which is generated matches the internal table.

We will cover the basics of header definition here. For more information on GEOS headers, refer to *The Official GEOS Programmer's Reference Guide.*

Most headers you create will begin with the following:

```
.word   0
.byte   3
.byte   21
```

These are standard values for the next block, icon width, and icon height, respectively.

Following these lines is the icon image. This must be bitmapped image data. The icon image is the picture which will appear in the deskTop directory window. If the icon image is more than 64 bytes, the remainder will be ignored; if the image is less than 64 bytes, it will be padded with zeros.

C64type is a Commodore file type. For GEOS applications, this will be $83.

GEOtype is the GEOS file type. If you **.include** the **constants** file, you can use the equated names, such as APPLICATION or DESK_ACC.

GEOstruct is the GEOS file structure type, meaning VLIR (0) or SEQUENTIAL (1).

FileStart is the program absolute load address. When your application is opened, GEOS will load it at this address. This should be the same value used in the linker's **.psect** directive. If you use a zero in this field, geoAssembler will use a default value of $400.

FileEnd is the program absolute end address. This value is only necessary for desk accessories, so GEOS can determine how much memory to save before overlaying the accessory code. This number should be $3ff for applications. If you use a zero in this field, geoAssembler will use a default value of $3ff.

InitProg is the address GEOS jumps to to begin execution of your application. If you use a zero in this field, geoAssembler will use a default value of $400.

filename is the ASCII name of the file (a string in double-quotes). If it is less than 20 characters, it must be padded with zeros (outside of the string) so that the total byte count is 20. The zero padding must occur within the same **.byte** statement.

File header blocks are exactly 256 bytes in length, but geoAssembler only requires that you give it ~~the first 97 bytes, up through the file name.~~ authorname (20 bytes) also required However, you may manually code the remaining fields with additional data, up to the full 256 bytes. geoAssembler will do no syntax checking beyond the 97th byte, though. If you submit less than 97 bytes, geoAssembler will generate an error; if you submit more than 97, but less than 256, geoAssembler will pad the remainder (up to 256) with zeros; if you submit more than 256 bytes, geoAssembler will generate an error.

The additional (optional) fields are are described fully in *The Official GEOS Programmer's Reference Guide.*

Example:

```
.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, with bit 7 set.
.byte   APPLICATION             ;Geos file type
.byte   SEQUENTIAL              ;Geos file structure type
.word   ProgStart               ;start address of program (where to load to)
.word   $3ff                    ;usually end address, but only needed for
                                ;desk accessories.
.word   ProgStart               ;init address of program (where to JMP to)
.byte   "SampleSeq  V1.0",0,0,0,$00
                                ;permanent filename: 12 characters,
                                ;followed by 4 character version number,
                                ;followed by 3 zeroes,
                                ;followed by 40/80 column flag.
.byte   "Eric E. Del Sesto ",0
                                ;twenty character author name
                                (including null)
;end of header section which is checked for accuracy
.block  160-117                 ;skip 43 bytes...
.byte   "This is the GeoProgrammer sample "
.byte   "sequential GEOS application.",0
.endh
```

Internal Variables

geoAssembler maintains three internal variables which you can use in your assembly source code:

picH	most recent icon's height
picW	most recent icon's width
Pass1	assembly on pass one or pass two

picH & picW

When geoAssembler encounters a graphic image in your source code, it will converts it into compacted bitmap data data and inserts directly into the object code, as if it was generated with **.byte** data directives. At this time, it also sets two internal variables: **picH** and **picW**. **picH** is the graphic image height in scanlines and **picW** is its width in bytes. Although the width and height of the most recent image remain in effect until a subsequent image definition, it is best to assign them to permanent equates immediately after the image:

```
Icon1 Picture:                  ;assembler will place co
                                ;here for this picture:

                                      Icon

ICON_1_WIDTH  = picW            ;store bitmap size values
ICON_1_HEIGHT = picH            ;table on pass 2. (picW a
                                ;the assembler.)
```

For more information on pasting images into your geoWrite source files, refer to "Including Icons (Graphics) in Your Source File" in Chapter 4.

Pass1

geoAssembler is a two-pass assembler. On the first pass it establishes values for labels and equates, increments section counters, and defines macros; on the second pass it resolves forward and backward references. Because no new information is presented in equates and macro definitions on the second pass, significant disk and file processing time can be saved by eliminating this redundancy. For this purpose, geoAssembler maintains an internal variable called **Pass1**. At the beginning of the first pass, **Pass1** is set to a logical true; at the beginning of the second pass, **Pass1** is set to a logical false. You can use the **Pass1** variable in a conditional assembly expression to exclude equates and macros from the assembly on the second pass. This can usually realize a 10% to 20% improvement on assembly time.

Example:

```
.if     Pass1     ;only include equ's and macros on 1st pass
        .include myEquates
        .include myMacros
        .include geosSym
.endif
```

IMPORTANT: Only use the **Pass1** variable to exclude equates and macro definitions from your assemblies. Using **Pass1** in any other context can cause symbols to evaluate differently on each pass. geoAssembler has no facility to detect these "phase" errors, and the results are unpredictable. Also: it is best to only use the **Pass1** facility when you are sure there are no errors in your include files. If there are errors in your include file and the files are not processed during the second pass, you will get a "hidden error" error message. If you should get a hidden error, remove the **Pass1** conditional and reassemble. The offending line(s) will then be flagged correctly in the error file. Once you have corrected the error, you can again use the **Pass1** conditional.

Chapter 6: geoLinker Reference

Chapter 6 acts as a complete reference for geoLinker, including the linker command file. Although this is primarily a reference chapter, it would be a good idea to read it through completely at least once. For information on using geoLinker from the GEOS deskTop, refer to "Running geoLinker" in Chapter 4.

The Link Process

geoAssembler generates .rel (relocatable object) files which consist of three main elements:

relocatable 6502 machine code
unresolved expressions
global labels and equates

geoLinker takes one or more .rel files and converts them into a runnable application.

When geoAssembler encounters an undefined symbol, a symbol which is not defined in the current source file, it assumes that it is an *external* symbol. Valid expressions which use external symbols get passed to geoLinker for resolution.

When you link a number of .rel files together, geoLinker matches-up external references from one file with the global symbols within other files, thereby resolving any valid cross-references.

During the link process, geoLinker also establishes fixed absolute addresses for the program code (**.psect**) sections and unitialized data space (**.ramsect**) sections. It then converts all the relocatable machine code into absolute machine code which is runnable in the GEOS environment.

If geoLinker is able to resolve all external references, it produces a runnable application file, complete with a proper GEOS header block, a .dbg symbol table for use with geoDebugger, and an optional .sym viewable symbol table.

Linker Overview

Command File

geoLinker needs a lot of information in order to integrate a group of .rel files into a GEOS application. Besides specifying the linkable modules, you can provide an output file name, a customized GEOS header, absolute psect and ramsect addresses, even VLIR overlay modules. All this information is specified in a linker command file. Like geoAssembler source files, linker command files are created in geoWrite.

Sequential and VLIR Applications

geoLinker is capable of generating both sequential and VLIR type application files. Sequential applications consist of one contiguous main module, which is loaded entirely into memory when you run the application. VLIR (Variable Length Indexed Record) applications consist of one resident module, very similar to a sequential file, which is loaded in when you run the application and any number of overlay modules, modules which are loaded into memory as they are needed.

Standard Commodore Applications

geoLinker can also generate standard Commodore application files for running outside of the GEOS environment with the **.cbm** linker directive. A standard Commodore application is much like a sequential GEOS application without a GEOS file header. For more information on standard Commodore applications, refer to the *Commodore 64 Programmer's Reference Guide.*

Header and Output File

geoLinker allows you to specify a GEOS file header and an output file in the linker command file. However, if you do not specify either or both, geoLinker will use a default. The default header uses a special test icon with a load and execution address which points to the first byte of the psect section. The default file name is **test.**

Psect and Ramsect Addresses

All psect and most ramsect sections are relocatable — they are given absolute addresses within the Commodore's memory space at link time. If you do not specify a particular psect address, geoLinker will default to $400. If you do not specify a particular ramsect address, geoLinker will append the ramsect section to the last byte of your psect section (or module for a VLIR appliction).

The Linker Command File

Any link operation must use a linker command file. Linker command files are created in geoWrite and are similar to geoAssembler source files — they consist of linker directives, file names, and comments.

Using geoWrite to Create Link Command Files

geoLinker command files must be in geoWrite format. You create them in much the same way you create you geoAssembler source code. For more information on using geoWrite, refer to "Creating geoAssembler Source Code" in Chapter 4 and the geoWrite section of your *GEOS User's Guide*.

NOTE: A linker command file cannot exceed one geoWrite page. Anything beyond the first page of text will be ignored.

Comments

Just as in geoAssembler, you may include comments in your linker command file with a semicolon (;). geoLinker will ignore text from a semicolon to the end of the line. Unless the semicolon is the first item on the line, it must be preceded by at least one space.

Directives

geoLinker has a small set of directives which allow you to control and specify different linker actions:

.output	specify output file name
.header	specify header .rel file name
.psect	set psect absolute address
.ramsect	set ramsect absolute address
.seq	start sequential application link
.vlir	start vlir application link
.mod	begin vlir overlay module
.cbm	start standard Commodore application link

Linker directives are not case-significant: you may type them in upper- or lower-case, or any mixture thereof.

Filenames

If an item or a line is not a comment nor a directive, geoLinker assumes it is a .rel relocatable object file. Files to link can only be specified on lines after a **..seq, .vlir, .mod,** or **.cbm** directive. They require no special syntax, except that there can only be one file name per line. File names are case-significant.

If you have two disk drives (one can be a RAMdisk), geoLinker will automatically search both for the desired file, starting with the same disk as the linker command file.

Expressions

geoLinker uses the same expression evaluator as geoAssembler. This allows you to include the same types of expressions within your linker command file as you would in your assembly source code. You can even use expressions which contain symbols equated in one of the assembly files. For example, rather than

```
.ramsect $4100
```

you might want to do something like

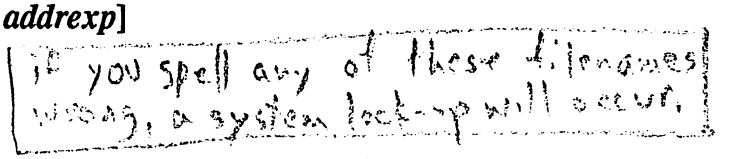
```
.ramsect buffer_start+100
```

which is valid, assuming **buffer_start** is equated in one of the .rel files. For more information on expression evaluation, refer to "Expressions" in Chapter 5.

Sequential Application Link

A sequential application uses a linker command file which follows this basic pattern:

```
[.output      filename]
[.header      filename.rel]
 .seq
[.psect       addrexp]
[.ramsect     addrexp]
 filename.rel
{filename.rel}
```



filename is a valid file name and *addrexp* is an expression which evaluates to an absolute address within the Commodore's memory space. As indicated by the bracketed sections, most of the contents are optional. The simplest linker command file, one which uses all the defaults, would look like this:

```
.seq
filename.rel
```

This would link one .rel file into a sequential application. It would use the default header, the default addresses, and the default application file name of **test**. Here is an example of a more complex sequential link file:

```
    .output    myprogram
    .header    myheader.rel
    .seq       ;this is a sequential app.
    .psect     prog_addr+$42e          ;program start
    .ramsect   $3000                   ;unitialized data start
;--- link all these files together ---
    myinit.rel
    mymain.rel
    mydata.rel
    mytable.rel
```

This linker command file will generate a sequential application called **myprogram**, using a header **myheader.rel**, and the assembled files **myinit.rel**, **mymain.rel**, **mydata.rel**, and **mytable.rel**. The .rel files will be relocated and appended to each other, one after the other, in the order they are listed in the linker command file. The program will be given an absolute address at **prog_addr$42e** (**prog_addr** must be equated in one of the .rel files) and an absolute ramsect address of $3000.

VLIR Application Link

A VLIR application requires a more complex linker command file to manage the overlay modules. The linker command file for a VLIR application follows this basic pattern:

```
[.output      filename]
[.header      filename.rel]
 .vlir
[.psect       addrexp]
[.ramsect     addrexp]
 filename.rel
{filename.rel}
.mod          exp
[.psect       addrexp]
[.ramsect     addrexp]
 filename.rel
{filename.rel}
```

Although a VLIR linker command file resembles a sequential linker command file, you will immediately notice the addition of the **.mod** overlay module directive. It might help to think of a VLIR application as a series of sequential applications merged into one program file. The main, or *resident*, module follows the **.vlir** directive. It can have it's own **.psect** and **.ramsect** and is made up of one or more .rel files. Each overlay module also has a unique number (indicated by *exp* above) and it's own separate **.psect** and **.ramsect,** in addition to it's own constituent .rel files.

Example:

```
      .output     myvlir
      .header     vlirhead.rel
;--- resident module ---
      .vlir
      .psect      $1000
      .ramsect    $4500
      init.rel
      dispatch.rel
      menus.rel
;--- overlay ---
      .mod        1          ;module #1
      .psect      swap_addr
      cutpaste.rel
;--- overlay ---
      .mod        2          ;module #2
      .psect      swap_addr
      rubbox.rel
      fill.rel
;--- overlay ---
      .mod        3          ;module #3
      .psect      swap2_addr
      io.rel
```

This linker command file would generate a vlir application called **myvlir,** using a header **vlirhead.rel.** The resident module consists of three files: **init.rel, dispatch.rel,** and **menus.rel.** There are also three overlay modules, numbered one through three, each with its own absolute address.

Cross-reference Resolution

geoLinker has some special features and limitations which affect the way it resolves cross-references.

How geoLinker Resolves Cross-references

When you assemble a file, geoAssembler assumes that any undefined symbol used in an expression is an external reference and sends the entire expression, unevaluated, to the .rel file. geoLinker will attempt to resolve this expression with global symbols from the other .rel files.

Global Label Conflicts

It is often the case that two or more .rel files will use identical symbols for unrelated labels or equates. When these files are linked, geoLinker will encounter these conflicting symbols. Ideally, the programmer would keep these symbols from the link stage by using the **.noglbl** and **.noeqin** directives. However, this is seldom practical. As a result, global labels frequently have duplicates during the link stage. geoLinker, however, will not flag these as duplicate label errors, assuming the conflict was unintentional, unless another .rel file tries to externally reference one of the symbols, in which case geoLinker has no way of deciding which one is desired. An error is generated.

NOTE: If a symbol can be resolved during the assembly stage, it will be and no external reference will be generated. Therefore, a routine which is internal to an object module will take precedence over a routine with the same name which is external to the module.

VLIR Overlay Module Refernces

A VLIR file typicvally has one resident module and many overlay modules. geoLinker links each module (whether resident or overlay) as if they were entirely independent sequential applications with one exception: an overlay module can reference symbols in the resident module. However the resident module is unable to access symbols in an overlay module and an overlay module cannot access symbols in other overlay modules.

This would seem to defeat the whole purpose of having an overlay linker. Fortunately, this limitation of symbol scope can be overcome by the use of jump tables. A VLIR jump table can be built at the beginning of each overlay module, and then a constants file can be used to index into this jump table. This is how Berkeley Softworks manages overlay modules in their VLIR applications. (For an example of an overlay jump table, refer to the sample VLIR application on your geoProgrammer disk.)

Link Directive Reference

Directive: **.output**

Purpose: Specifies an output file name for the application and a base file name for its .sym, .dbg, and .err files.

Usage: **.output** *filename*

Note: *filename* is a valid file name.

The **.output** directive allows you to specify an output file name for use in any files which geoLinker generates during the current link. If you do not have a **.output** directive, geoLinker will use the name **test**. You should not specify an extender in the file name; geoLinker will append a .sym, .dbg, or a .err where appropriate.

The **.output** directive, if used, must be the first directive in the linker command file.

Example:

```
.output    myapp
```

This would signal geoLinker to use the name **myapp** for the name of the linked application and as the base file name for any associated files **(myapp.sym, myapp.dbg, myapp.err)**.

Directvie: **.header**

Purpose: Specifies a previously assembled .rel file to be used to generate the GEOS file header.

Usage: **.header** ***filename*.rel**

Note: *filename* is a valid file name. You must manually append the .rel extender.

The **.header** directive allows you to specify a .rel file which contains data for the GEOS file header, most likely created with geoAssembler's **.header/.endh** directives.

geoLinker expects the header file to contain exactly 256 bytes of object code. If you create the header with the geoAssembler **.header** directive, 256 bytes will always be generated. Otherwise, you will need conform to this count manually. If the header file contains more or less than 256 bytes of object code, an error will be generated.

If you omit the **.header** directive, a default header will be generated with the appropriate sequential or VLIR flags set. The default header uses a load and execution address which points to the first byte of the psect section (resident module) of your code. The default header cannot be used to generate desk accessories.

The **.header** directive must appear after any **.output** directive but before the **.seq**, or **.vlir** directive. **.header** is invalid with the **.cbm** directive.

Example:

```
.header    seqhead.rel
```

This would signal geoLinker to use the file **seqhead.rel** as data for the application's GEOS file header.

For more information on the GEOS file header, refer to "Header Directives" in Chapter 5 of this manual. Also refer to *The Official GEOS Programmer's Reference Guide.*

Directive: **.psect**

Purpose: Establish an absolute address for program code and data (psect) section.

Usage: **.psect** ***addrexp***

Note: *addrexp* is an expression which evaluates to an absolute address.

geoAssembler does not resolve absolute addresses of psect sections until link-time. For this reason, you can use the **.psect** link directive to specify an absolute address for your program code and data. If you omit the **.psect** directive, geoLinker will use a default value of $400.

The **.psect** directive is only valid after a **.seq**, **.vlir**, **.mod**, or **.cbm**, and it must appear before the associated .rel file names. In a VLIR link, **.psect** will only affect the most recent resident or overlay module.

Example:

```
.vlir
.psect      $500        ;resident mod $500
app.rel
text.rel
.mod        1
.psect      $1000       ;this overlay at $1000
over1.rel
.mod        5
.psect      $1000       ;this one at $1000, too
.over5.rel
.mod        3
;                       ;but use default ($400) here
over3.rel
```

Directive: **.ramsect**

Purpose: Establish an absolute address for relative unitialized data (ramsect) sections.

Usage: **.ramsect** ***addrexp***

Note: *addrexp* is an expression which evaluates to an absolute address.

If you want, you can let geoLinker resolve the absolute addresses of your ramsect sections by not supplying an absolute address with your geoAssembler **.ramsect** directives. You can use the geoLinker **.ramsect** directive to specify an absolute address for your these ramsect sections. If you omit the **.ramsect** directive, geoLinker will automatically place your ramsect section immediately after the end of the psect section. If a VLIR overlay module does not have a **.ramsect**, the ramsect section will be placed after the overlay module's psect section, not the resident module's.

The **.ramsect** directive is valid after a **.seq**, **.vlir**, **.mod**, or **.cbm** directive. All the relative ramsect sections within that module will be appended and resolved based on that address.

Example:

```
.vlir
.psect      $500        ;resident mod $500
app.rel                 ;placing ramsect after psect section
text.rel
.mod        1
.psect      $1000       ;this overlay at $1000
.ramsect $2000          ;this module's ramsect at $2000
over1.rel
over1a.rel
over1b.rel
.mod        5
.psect      $1000       ;this one at $1000, too
;ramsect omitted: ramsect section will be placed right
;after psect section by linker.
typeset.rel
```

Directive: **.seq**

Purpose: Alerts geoLinker that this is a sequential application and that all the file names following should be linked into one main, entirely resident program.

Usage: **.seq**

Note: Takes no parameters.

In order to generate a sequential application, you must place this directive before any **.psect**, **.ramsect**, or linkable file names. It signals geoLinker to create a sequential application.

A linker command file must have at least one **.seq**, **.vlir**, or **.cbm** directive, but not more. The only directives which can appear before the **.seq** are **.output** and **.header**. The **.mod** directive cannot be used with **.seq**.

If the **.header** directive is omitted, geoLinker will generate a default sequential header.

Example:

```
.seq        ;generate a sequential GEOS application
init.rel
main.rel
subrtn.rel
data.rel
```

Directive: **.vlir**

Purpose: Alerts geoLinker that this is a VLIR (Variable Length Indexed Record) application and that the following (up to a **.mod** directive) are part of the resident module.

Usage: **.vlir**

Note: Takes no parameters.

In order to generate a VLIR application, you must place this directive before any **.psect, .ramsect, .mod,** or linkable file names. It signals geoLinker to create a VLIR application.

A linker command file must have at least one **.seq, .vlir,** or **.cbm** directive, but not more. The only directives which can appear before the **.vlir** are **.output** and **.header.**

.psect and **.ramsect** directives for the resident module must come before any .rel file names. **.psect** and **.ramsect** directives immediately following a **.vlir** directive will only affect the resident module, they will not affect any overlay modules created with the **.mod** directive.

Example:

```
     .vlir        ;generate aVLIR application
;*** resident module ***
     init.rel
     main.rel
     subrtn.rel
     data.rel
;*** overlay module ***
     .mod      1
     over.rel
     data2.rel
```

The above will generate a VLIR application with one resident module and one overlay module, using all the defaults.

For more information on VLIR applications, refer to "VLIR Application Link" in this chapter. See also **.mod** in this chapter.

Directive: **.mod**

Purpose: Begin overlay module.

Usage: **.mod** *exp*

Note: *exp* is an expression which evaluates to a number between 1 and 126.

VLIR applications have one resident module and up to 20 overlay modules. You tell geoLinker to begin a new overlay module with the **.mod** directive followed by a module number (1 through 126). The module number becomes the record number within the VLIR file. Keep in mind that all loading and swapping of overlay modules is a function of your program; the **.mod** directive merely allows you to resolve a group of .rel files together into one VLIR record.

At the beginning of an overlay module, the psect and ramsect counters are reset to their defaults. If you use a **.psect** directive, it must appear after the **.mod** directive, but before any .rel files. If **.psect** is not specified, geoLinker will default to $400; if no **.ramsect** is specified, the ramsect section will be appended directly after the last psect byte in the overlay module.

NOTE: You may choose any module number you desire. You need not begin with module one, and you need not use a contiguous number system. The only restriction is that the total number of modules must not exceed 20.

Example:

```
     .vlir        ;generate aVLIR application
;*** resident module ***
     init.rel
     main.rel
     subrtn.rel
     data.rel
;*** overlay module #1 ***
     .mod 1
     over.rel
```

```
    data2.rel
;*** overlay module #113 ***
    .mod 113
    cram.rel
    data3.rel
```

The above will generate a VLIR application with one resident module and two overlay modules, using all the defaults.

For more information on VLIR applications, refer to "VLIR Application Link" in this chapter. See also **.vlir** in this chapter. For more information on developing your own overlay management routines, refer to the sample VLIR application on your geoProgrammer disk.

Directive: **.cbm**

Purpose: Alerts geoLinker that this is a standard Commodore application and that all the file names following should be linked into one main, entirely resident program.

Usage: **.cbm**

Note: Takes no parameters.

In order to generate a standard Commodore application, you must place this directive before any **.psect**, **.ramsect**, or linkable file names. It signals geoLinker to create an application file comparable to the application files generated by other assemblers. A standard Commodore application cannot be run from the GEOS deskTop.

A linker command file must have at least one **.seq**, **.vlir**, or **.cbm** directive, but not more. The only directives which can appear before the **.cbm** are **.output** and **.header**. The **.mod** directive cannot be used with **.cbm**.

geoLinker will *not* generate a GEOS file header for the application. If you use the **.header** directive, an error will be generated. The file will appear on the deskTop as a C= folder icon. If you want the application to be runnable from the GEOS deskTop, you can convert it to GEOS format with the Icon Editor (included with DESKPACK1). This will allow the file to be accessed from the GEOS deskTop, although it will not be a true GEOS application.

Example:

```
.cbm        ;generate a standard Commodore application
init.rel
main.rel
subrtn.rel
data.rel
```

Chapter 7: geoDebugger Usage and Tutorial

This chapter introduces the major features of geoDebugger as well as overviewing how to run and use the debugger. It begins by describing the concept of debugging and finishes with a short tutorial covering the basic features of geoDebugger. This chapter *does not* cover aspects of the debugger in exhaustive detail (refer to Chapters 8 and 9 for more complete coverage).

What is a Debugger?

When a program doesn't work, it is said to have *bugs*. Although the original meaning of the term is lost in antiquity, it still offers a rather vivid metaphor: you can almost see the little creatures crawling between opcodes, chewing on operands, and flipping bits in your data space. But the image is also misleading because debugging a program is seldom as simple as setting off a room fogger. It's more like tuning a car. It's an interactive process where you monitor the internal states of your program, looking for a bad value, a misused instruction, or a call to the wrong subroutine. As the car mechanic has tools for monitoring firing times, engine speed, and valve pressure, the programmer has tools for monitoring register states, stack usage, and variable space. The programmer's tool is the debugger.

geoDebugger Features

geoDebugger is a software version of a professional hardware debugging system used by Berkeley Softworks for in-house development. Outside of a few features which require an in-circuit emulator (a hardware device which replaces the 6502 microprocessor), the full functionality of this debugger has been preserved. You will likely find features, commands, and capabilities not found in any other software debugger.

geoDebugger offers a complete repetoire of commands to monitor your program and memory, giving you full access to the Commodore memory space and the 6502 registers. You can disassemble, modify, and run your program interactively, setting breakpoints, displaying and changing register

values, even loading and saving disk blocks. There is also a dynamic macro language which allows you automate common operations and customize the debugger to your liking.

Dual Displays

geoDebugger maintains a text screen which is entirely independent of the current application's screen. This allows you to have a full display of debugger information without disrupting the application's display. When you single step through your application's code, it will actually update its own display rather than corrupting the debugger screen. From within geoDebugger you can view the application's display by pressing F7.

Hot Key

geoDebugger traps the NMI (non-maskable Interrupt) generated by the RESTORE key. You can enter geoDebugger any time your application is running by pressing RESTORE. Because of the way the RESTORE key connects to the NMI line, you may have to press it more than once before it responds.

Symbolic Debugging

When you create an application, geoLinker generates a .dbg debugger symbol file. geoDebugger will load this file and will uses your symbols (global labels, and equates made with the == directive) during disassembly and memory display. You can also use these symbols within expressions and as command arguments.

Breakpoints

geoDebugger implements a user-defined breakpoint facility which allows you to mark up to eight places in your program as breakpoints. When these instructions are encountered (but before they are executed), your application is stopped and geoDebugger is given control. Breakpoints allow you to stop your program at specific point so you can examine registers and variables or perform some other debugging operation.

Expressions

geoDebugger includes a complete expression evaluator, much like the one in geoAssembler, with the addition of special symbols and operators appropriate in the debugging environment. The geoDebugger expression evaluator allows values from memory, processor registers, and special debugger variables to be included in the expression.

Open Modes

In addition to regular commands, geoDebugger offers special commands which place you into an interactive "open" mode. Open modes allow you to dynamically view and alter machine code, data, or the processor registers.

Debugger Macros

geoDebugger includes a full macro language which allows you to automate multiple keystrokes and common debugger functions. Macros offer a means to customize the debugger with your own commands. There is also a special AUTOEXEC macro which will be executed when you run geoDebugger. It can be used to automatically configure the debugger each time you run it.

Super-debugger and Mini-debugger

geoDebugger automatically configures itself as either a super-debugger or a mini-debugger.

The super-debugger is designed for professional development. In order to take full advantage of its features, you must have a RAM-expansion unit. The super-debugger is a large program and it "hides" within the 64K of system space in the RAM-expansion (it won't disrupt any files), leaving the memory inside the Commodore available for GEOS and your application. Not only does this allow you to develop applications which use the entire memory space, it also makes geoDebugger virtually transparent to the application. If you are serious about programming, this functionality is well-worth the price of a RAM-expansion unit.

Without a RAM-expansion unit, geoDebugger configures itself as a mini-debugger. The mini-debugger is a scaled down version of the super-debugger which resides within the Commodore memory space along with your program. It is missing many of the features of the full debugger, such as symbols and macros, and consumes about 8K of space, but it is still functional and valuable if you don't have access to a RAM-expansion unit. If you have a RAM-expansion unit, you can force the mini-debugger configuration by holding down the RUN/STOP key while geoDebugger is loading.

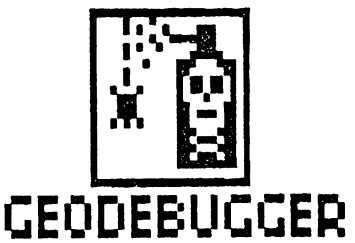
Running the Super-debugger

Assuming you have a RAM-expansion unit, there are two ways to run geoDebugger from the GEOS deskTop and have it configure itself as a super-debugger: either by opening geoDebugger by itself or by opening a .dbg debugger symbol file.

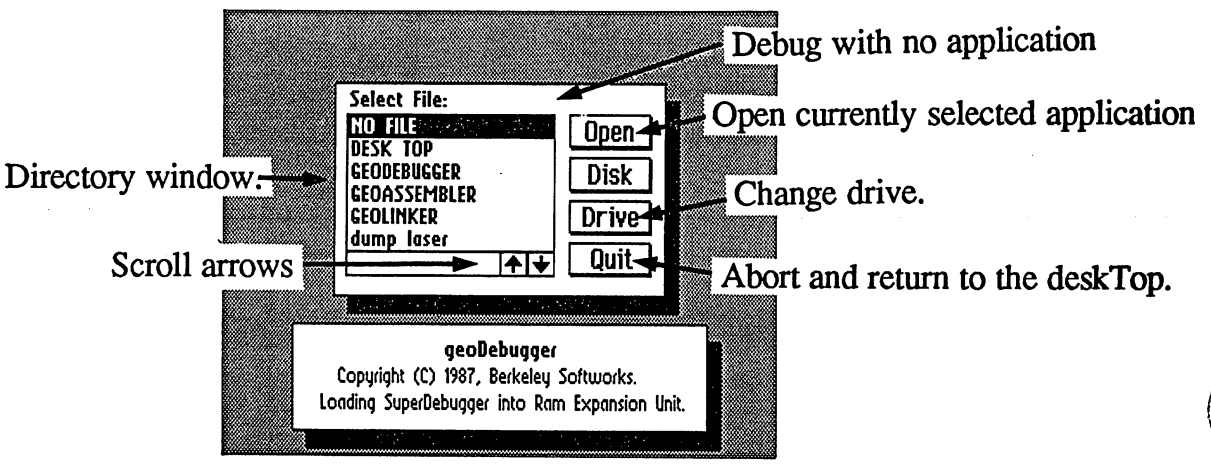
Running the Super-debugger by Itself:

To run the super-debugger by itself, follow these stpes:

1: Double-click on or open the GEODEBUGGER file.



2: After the super-debugger loads and initializes, you should see the following dialog box:



geoDebugger is asking for the name of an application to debug.

The contents of the current drive (the drive from which you ran geoDebugger) will appear in the directory window. If more items exist than can fit in the window, click on the scroll arrows to move through the directory.

The top entry **NO FILE** is not actually a file on your disk. If you open this entry, no application will be loaded. The **NO FILE** selection is useful if you want to test a feature of the debugger, quickly try a programming idea, test a GEOS routine, or simply experiment with your computer.

If you decide you do not want to use the debugger at this time, click on the **Quit** icon to abort and return to the deskTop.

To debug an application from a different drive (for example, a RAM-expansion unit or a second floppy drive), click on the **Drive** icon; the directory of the other drive will be displayed in the directory window.

The **Disk** icon allows you to view the contents of a different disk. To view the contents of a different disk, insert a new disk into the current drive and click on the Disk icon. The directory will be updated to show the contents of the new disk. The **Disk** icon will have no effect with a RAM-expansion Unit.

3: Select the application file you want to debug (or **NO FILE**) by clicking on the name. Then click on the **Open** icon to load the application into memory. The super-debugger will attempt to load the application as well as a .dbg debugger symbol file and .dbm debugger macro file which have the same basic file name as the application. If this .dbm file is not found, the super-debugger will look for a **default.dbm** debugger macro file.

Running the Super-debugger by Opening a Symbol File

You can run the super-debugger and have it automatically load the application to debug along with the appropriate symbols and macros by opening the .dbg debugger symbol file created by geoLinker.

Double-click on or open a .dbg debugger symbol file associated with the application you want to debug. geoDebugger will run and automatically load the symbol table. geoDebugger will also try to load the application

which owns the symbol table as well as a .dbm macro file associated with the application. For example: if the **SamSeq.dbg** symbol file is opened, geoDebugger will load in those symbols as well as the **SamSeq** application file and the **SamSeq.dbm** debugger macro file.

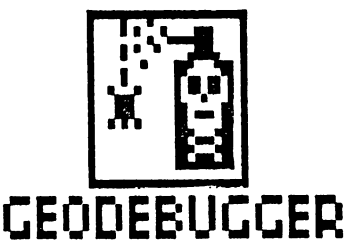
If geoDebugger cannot find a debugger macro file that is associated with the application, it will look for a **default.dbm** file on the same disk. If this macro file exists, it will be loaded.

Running the Mini-debugger

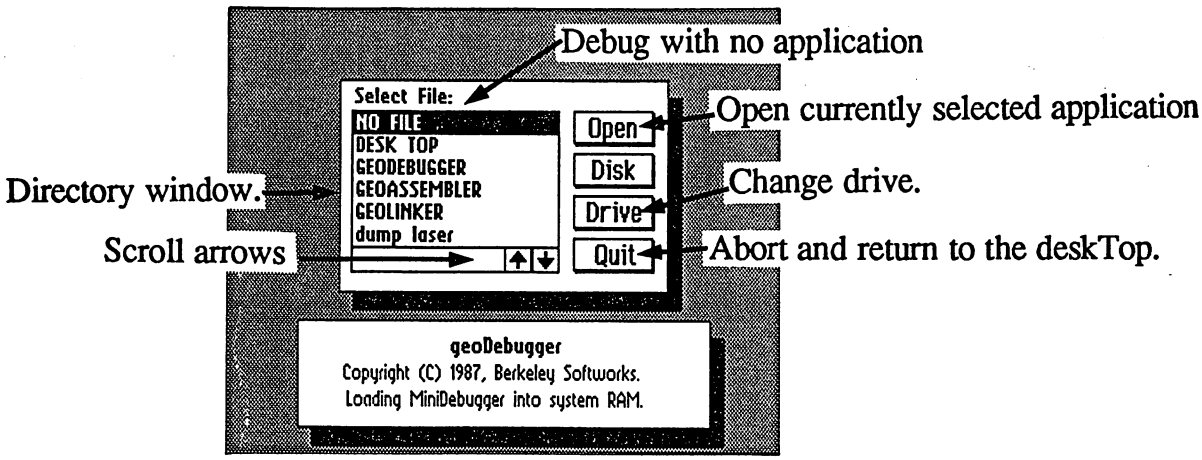
If you do not have a RAM-expansion unit, geoDebugger will automatically configure itself as a mini-debugger. If you do have a RAM-expansion unit, you can force this configuration by holding down the RUN/STOP key while geoDebugger is loading.

To run the mini-debugger, follow these steps:

1: Double-click on or open the GEODEBUGGER file.



2: After the mini-debugger loads and initializes, you should see the following dialog box:



geoDebugger is asking for the name of an application to debug.

The contents of the current drive (the drive from which you ran geoDebugger) will appear in the directory window. If more items exist than can fit in the window, click on the scroll arrows to move through the directory.

The top entry **NO FILE** is not actually a file on your disk. If you open this entry, no application will be loaded. The **NO FILE** selection is useful if you want to test a feature of the debugger, quickly try a programming idea, test a GEOS routine, or simply experiment with your computer.

If you decide you do not want to use the debugger at this time, click on the **Quit** icon to abort and return to the deskTop.

To debug an application from a different drive (for example, a second floppy drive), click on the **Drive** icon; the directory of the other drive will be displayed in the directory window.

The **Disk** icon allows you to view the contents of a different disk. To view the contents of a different disk, insert a new disk into the current drive and click on the Disk icon. The directory will be updated to show the contents of the new disk.

3: Select the application file you want to debug (or **NO FILE**) by clicking on the name. Then click on the **Open** icon to load the application into memory. The mini debugger will load the application but will not load any associated symbol or macro files because it does not support symbols or macros.

As with the super-debugger, the mini-debugger can be automatically loaded by opening the .dbg symbol file associated with the application you want to debug. However, because the mini-debugger does not support symbols or macros, a .dbg debugger symbol file and the .dbm debugger macro file will not be loaded.

Sample Super-debugger Session

This section is a hands-on tutorail. It is designed to familiarize you with the the super-debugger environment by using the super-debugger with the sample application. If you do not have a RAM-expansion unit, you cannot run the super-debugger; refer to "Sample Mini-debugger Session" in this chapter. If you have not yet created the **SampleSeq** application, refer to "Creating a Sample Application" in Chapter 4.

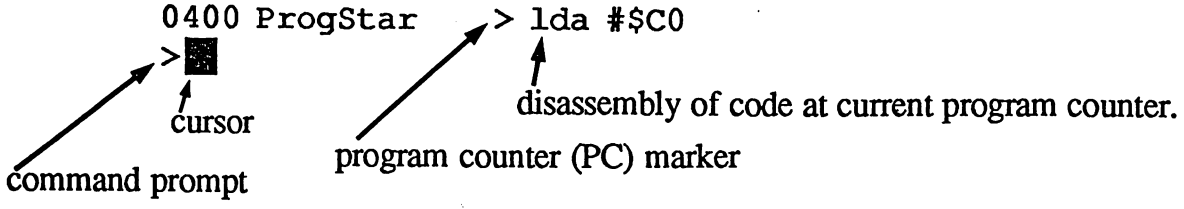
Before running the super-debugger, you will need a disk with the following files on it:

GEODEBUGGER	*debugger*
SampleSeq	*sample sequential application*
SampleSeq.dbg	*symbols for the sequential application*
SampleSeq.dbm	*sample debugger macro file*

Running the Super-debugger with the Sample Application

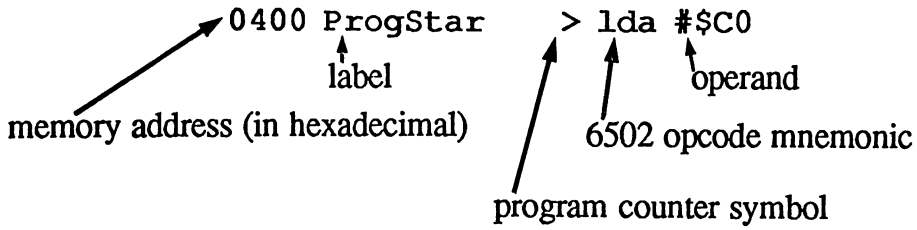
Since we want to debug the sample application, we can have geoDebugger load the application, its symbols, and its macro file by opening or double clicking on the symbol file. To do this, open **SampleSeq.dbg**, which is the symbol file for the sequential application. geoDebugger will run and automatically configure itself as the super-debugger and load everything. The screen will enter text mode and you will see the following display:

```
geoDebugger

Copyright (C) 1987 Berkeley Softworks

Program file: SampleSeq loaded.
Loading macro definitions.
Loading symbol definitions.

0400 ProgStar     > lda #$C0
>
```



After loading the application, macros, and symbols, geoDebugger places the program counter (the 6502 register which points to the *next* instruction to be executed) at the application's start address and disassembles the instruction at that address. *Disassembly* the reverse process of assembly; geoDebugger looks into memory and translates the binary codes into assembly-language mnemonics:

```
0400 ProgStar     > lda #$C0
```



After geoDebugger has disassembled this line, it will display the command prompt. The geoDebugger command prompt is a greater-than (>) symbol in the leftmost column of the screen. Whenever this prompt is displayed, geoDebugger is idle, awaiting a command. To enter a command, you type the command along with any parameters and press the RETURN key. If you make a mistake while typing, you can back up one character at a time by pressing DEL or you can clear the entire entry by pressing ←.

To get a full screen disassembly of the sample application, enter the **dis** (disassemble) command (remember to press RETURN). The following will be displayed:

```
0400 ProgStar     >       lda #$C0
0402 ProgStar+$02         sta dispBuff
0404 ProgStar+$04         lda #$04
0406 ProgStar+$06         sta r0h
0408 ProgStar+$08         lda #$28
040A ProgStar+$0A         sta r0l
040C ProgStar+$0C         jsr Graphics
040F ProgStar+$0F         lda #$04
0411 ProgStar+$11         sta r0h
0413 ProgStar+$13         lda #$33
0415 ProgStar+$15         sta r0l
0417 ProgStar+$17         lda #$00
0419 ProgStar+$19         jsr DoMenu
041C ProgStar+$1C         lda #$04
041E ProgStar+$1e         sta r0h
0420 ProgStar+$20         lda #$85
0422 ProgStar+$22         sta r0l
0424 ProgStar+$24         jsr DoIcons
0427 ProgStar+$27         rts
0428 ClearScr             ora r0l
```

This is a disassembly of the first 20 instructions in the **SampleSeq** application. The program counter is shown at **$0400** with the **>** symbol; the **lda #$C0** is the first instruction in the application.

At this point it would be useful to compare this disassembly to the actual **SamSeq** source code file. You will immediately notice some differences: any symbol which is longer than eight characters has been truncated to eight, hence **dispBufferOn** becomes **dispBuff**; macros have been expanded, hence all the **Loadw**'s and **LoadB**'s are shown as their constituent **lda**'s and **sta**'s; and all expressions have been evaluated.

But you don't want to see your source code. That's what a program listing is for. You want to look at the code which was actually generated.

Executing Some Code

The routine at **ProgStar** clears the screen, points GEOS to the menu and icon structures, and then does an **rts** to the GEOS **MainLoop**. The **jsr Graphics** (truncated from **GraphicsString**) clears the screen. The **jsr DoMenu** places the menus up. The **Jsr DoIcons** places the icon on the screen.

Enter the command **runto 419**. The screen will momentarily flash to the application screen and then back to the debugger screen. The following will be printed:

```
0419 ProgStar+$19 > jsr DoMenu
```

The **runto** command set a breakpoint at address $419 (we didn't need to type the $ because geoDebugger's default radix is hexadecimal) and began executing code beginning at the current location of the program counter, which was $400. Notice that the program counter is now at $419. This means that the instruction **jsr DoMenu** has not yet been executed, but is next on the list. However, all the code from $400 (**ProgStar**) to $419 (**ProgStar+$19**) was executed.

Watching the Menus Go Up

To view the current state of the applictation's screen display, press F7 . You will see a blank screen with the sprite pointer in the upper left corner. Press F7 again (or any other key) to return to the debugger screen.

Now enter the **t** (top-step) command. The top step will single-step through the current instruction and return control to the command prompt at the next instruction. In the case of **jsr DoMenu**, the top-step will execute the subroutine **DoMenu** at full-speed and return when the next instruction (**lda #$04**) is encountered. If we wanted to, we could have single-stepped through the **DoMenu** subroutine using the **s** (single-step) command. When the top-step returns, the next instruction, at the new location of the program counter, will be printed:

```
041C ProgStar+$1C > lda #$04
```

Now if you press F7 to view the application's GEOS screen, you will see the effects of the **DoMenu** subroutine: the menus have been drawn. Return to the debugger screen by pressing any other key.

We can now top-step through the next two instructions by pressing . twice. If . is typed as the first character on a line, the previous command (in this case **t**) will be executed again. Pressing it twice should yield the following display:

```
041E ProgStar+$1e > sta r0h
0420 ProgStar+$20 > lda #$85
```

The first press executed the **lda #$04** and displayed the next instruction to be executed (**sta r0h**), and the second press executed the **sta r0h** and displayed the next instruction to be executed (**lda #$85**).

Showing Registers

To view the current state of the processor's registers, enter the **r** (show registers) command:

```
Acc  X   Y    PC   SP NV-BDIZC  MemMap
$04 $00 $07 $0420 $FD 00100001 00110000
```

The two registers of interest here are the **Acc** (accumulator) and **PC** (program counter). The accumulator contains a $04, which is still around from the **lda #$04** at $41c, and the program counter is at $420.

Using One of the Sample Macros

Macros are invoked, or executed, just like commands. One of the sample macros which was automatically loaded when you ran the debugger is the **sr** macro. The **sr** macro single-steps and then shows the processor registers. It is a combination of the **s** (single-step) command and the **r** (show registers) command. Since the next instruction to execute is a **lda #$85**, the **sr** ought to single-step through the instruction, thereby loading a $85 into the accumulator, and then it should show the result of this command. Invoke the **sr** macro now. You should see the following:

```
0422 ProgStar+$22 > sta r0l

Acc  X   Y    PC   SP NV-BDIZC  MemMap
$85 $00 $07 $0422 $FD 00100001 00110000
-------------------------------------
```

The single-step executes the **lda #$85**, and then disassembles the next instruction at $422. The register display shows that the accumulator (**Acc**) register now contains a $85, the result of the **lda**.

Running the Code Full-speed

Now that we've examined the application from within the debugger, let's run it full-speed. Use the **go** command. The **go** command displays the application's screen and begins execution at the current value of the program counter.

You can now experiment with the sample application. Click on the icon or select menu items. The application is running full-speed and has no idea that the debugger is lurking in the background just waiting to be called up in case of an error. Be careful, though, do not select **Quit** from the **file** menu because the application will attempt to leave to the deskTop and will be stopped by geoDebugger.

Hot Key Entry Into geoDebugger

When you are done playing with the application, press RESTORE . This is the geoDebugger "hot key." Whenever you press it, geoDebugger will immediately take control. The applications screen will be replaced with the debugger screen and the following message will be printed:

```
*** Execution stopped ***
FDAB $FDAB       > bpl $FDA4
```

The current location of the program counter (the instruction which was about to be executed when you pressed RESTORE) will be disassembled. The actual address will most likely be different than above because it depends on what instruction was being executed when you pressed RESTORE .

Whenever you enter geoDebugger with the RESTORE key, there is always a chance that the program will be in the middle of interrupt code. This is not problematic in itself, but can wreak havoc with disk I/O and some GEOS applications. Unless you are sure of what you are doing, it is always a good idea to execute a **stopmain** command. Do this now. **stopmain** sets a breakpoint in a safe place in GEOS **MainLoop** and then returns to the program. Since most properly written GEOS applications will eventually return to **MainLoop**, the breakpoint will usually be encountered.

Modifying Program Data

One of the great benefits of a debugger is the ability to quickly modify an application and test the results. In geoDebugger it is very easy to modify

instructions and program data. Say, for example, we don't like the way our menus look and we would like to modify them. Looking at the source code, we see that the text for the **geos** menu is store at **GeosText**. Enter **m GeosText**. The **m** command means open memory for display and modification as data. **GeosText** is the address of the first location to open. You will see the following:

```
0461 GeosText          *    .byte $67
```

Callouts: Cursor

Because the **m** command is an open mode, all further keystrokes, until the open mode is exited, will be interpreted by the **m** command. This allows the data display and entry to be interactive.

The $67 is the first character of the **geos** text string for the menu. We can display this hex value as a character by pressing the ' (SHIFT + 7) character radix symbol. The display will automatically update:

```
0461 GeosText          *    .byte 'g
```

By pressing {up/dn} three times, we can move through memory to see the remaining characters:

```
0461 GeosText              .byte 'g
0462 GeosText+$01          .byte 'e
0463 GeosText+$02          .byte 'o
0464 GeosText+$03      *   .byte 's
```

Now that we've seen the data, we can modify it. Press SHIFT + ↕ three times to return back to address $461, where the **g** is:

```
0461 GeosText          *    .byte 'g
```

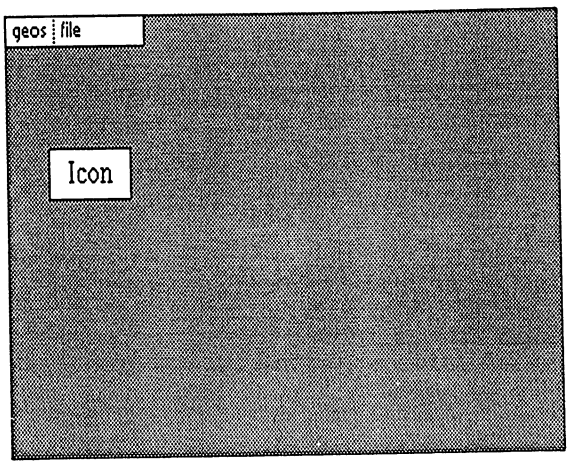
Press SPACE . The 'g will disappear and the cursor will be moved into the data field of the **.byte**. Type **"GEOS"** (including the surrounding double-quotes) and press RETURN . Since the text string is four characters (four bytes) long, it will be deposited across four bytes, overwriting the lowercase **geos**:

```
0461 GeosText           .byte 'G
0462 GeosText+$01       .byte 'E
0463 GeosText+$02       .byte 'O
0464 GeosText+$03       .byte 'S
```

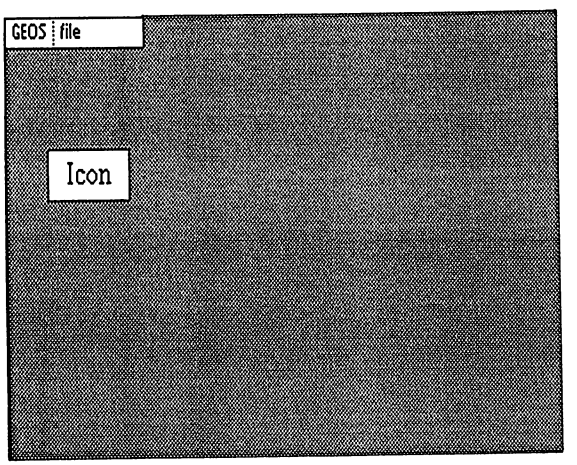
The data has now been modified in memory and control returned to the command prompt.

Testing the New Menu

To run the program and see the new menu text, enter the **go** command:



But notice that the menu text hasn't actually changed. This is because, although the text data in memory has been modifed, the menu needs to be redrawn by GEOS before this change will be reflected in the display. To force a redraw of the menu, select an item from under the **geos** menu:



As soon as the menu is redrawn, the new uppercase **GEOS** menu will appear.

And Now on to More Powerful Manipulations

Now that you have completed the sample session with the super-debugger, you should be beginning to feel comfortable with the debugging environment. You will find a complete discussion of the super-debugger command set in chapter 8.

Sample Mini-debugger Session

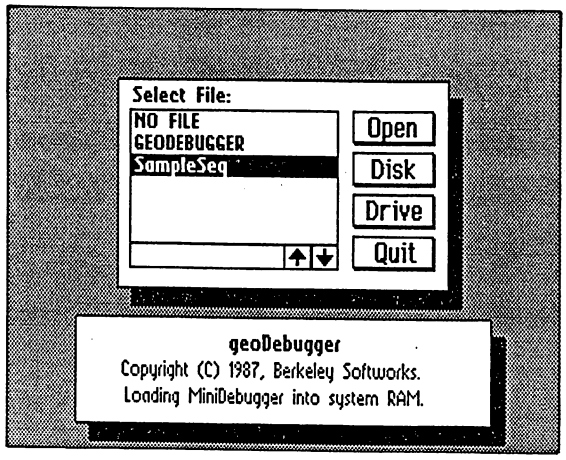
This section is designed to familiarize you with the the mini-debugger environment by using the mini-debugger with the sample application. Without a RAM-expansion unit, geoDebugger will automatically configure itself as a mini-debugger. If you have a RAM-expansion unit, you can configure geoDebugger as a mini-debugger by holding down the RUN/STOP key while the program is loading. If you have not yet created the **SampleSeq** application, refer to "Creating a Sample Application" in Chapter 4.

Before running the mini-debugger, you will need a disk with the following files on it:

GEODEBUGGER	*debugger*
SampleSeq	*sample sequential application*

Loading the Mini-debugger

Double-click on or open the GEODEBUGGER file from the deskTop. If you have a RAM-expansion unit, press and hold the RUN/STOP key while the debugger is loading. The screen will clear and a file-selection dialog will appear:



Click on the **SampleSeq** file to select it, then click on **Open.** The mini-debugger will load in the sample application, the screen will enter text mode and you will see the following display:

```
                geoDebugger

 Copyright ' 1987 Berkeley Softworks

Program file: SampleSeq loaded.

0400 A9 C0          > lda #$C0
>
```

disassembly of code at current program counter.

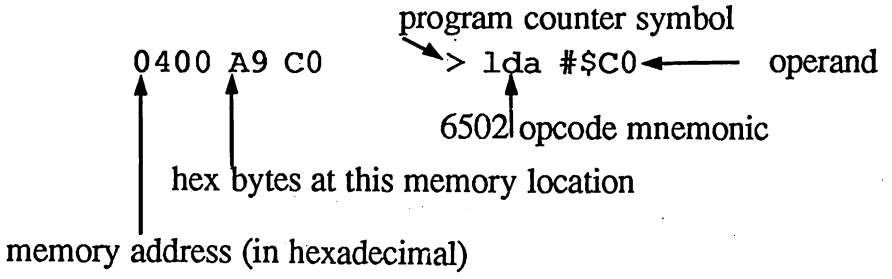
cursor

program counter (PC) marker

hex bytes of code at this location

command prompt

After loading the application, geoDebugger places the program counter (the 6502 register which points to the *next* instruction to be executed) at the application's start address and disassembles the instruction at that address. *Disassembly* the reverse process of assembly; geoDebugger looks into memory and translates the binary codes into assembly-language mnemonics:



After geoDebugger has disassembled this line, it will display the command prompt. The geoDebugger command prompt is a greater-than (>) symbol in the leftmost column of the screen. Whenever this prompt is displayed, geoDebugger is idle, awaiting a command. To enter a command, you type the command along with any parameters and press the RETURN key. If you make a mistake while typing, you can back up one character at a time by pressing DEL or you can clear the entire entry by pressing ← arrow}.

To view the code in memory as assembly langauge, enter the **a** command (remember to press RETURN). The following will be displayed:

```
0400 A9 C0          lda #C0
```

The **a** command tells the mini-debugger to open the current memory location for display and modification as assembly code. The cursor is placed over an asterisk symbol to indicate that we are now in an open mode and that subsequent kestrokes will be interpreted accordingly. To get a full screen disassembly of the sample application, press ↕ about 18 times until the screen fills with code (when the bottom of the screen is reached, the copyright information will scroll off the top):

```
0400 A9 C0        > lda #$C0
```

```
0402 85 2F          sta $2F
0404 A9 04          lda #$04
0406 85 03          sta $03
0408 A9 28          lda #$28
040A 85 02          sta $02
040C 20 36 C1       jsr $C136
040F A9 04          lda #$04
0411 85 03          sta $03
0413 A9 33          lda #$33
0415 85 02          sta $02
0417 A9 00          lda #$00
0419 20 51 C1       jsr $C151
041C A9 04          lda #$04
041E 85 03          sta $03
0420 A9 85          lda #$85
0422 85 02          sta $02
0424 20 5A C1       jsr $C15A
0427 60             rts
```

Press RETURN to leave the **a** open mode and return to the command prompt.

This text on the screen is a disassembly of the first 19 instructions in the **SampleSeq** application. The program counter is shown at **$0400** with the > symbol; the **lda #$C0** is the first instruction in the application.

At this point it would be useful to compare this disassembly to the actual **SamSeq** source code file. You will immediately notice some differences: any symbols are shown as their actual hexadecimal address, hence **dispBufferOn** becomes **$2F** and **DoIcons** becomes **$C15A**; macros have been expanded, hence all the **Loadw**'s and **LoadB**'s are shown as their constituent **lda**'s and **sta**'s; and all expressions have been evaluated and converted into hex values.

But you don't want to see your source code. That's what a program listing is for. You want to look at the code which was actually generated.

Executing Some Code

The routine at **$400** clears the screen, points GEOS to the menu and icon structures, and then does an **rts** to the GEOS **MainLoop**. The **jsr $C136 (jsr GraphicsString** in the source code) clears the screen. The **jsr $C151 (jsr DoMenu** in the source code) places the menus up. The

jsr $C15A (**jsr DoIcons** in the source code) places the icon on the screen.

Enter the command **sb 419**. The **sb** command (set breakpoint) will set a user-defined breakpoint at address $419 (we didn't need to type the $ because the mini-debugger operates entirely in hexadecimal). The following will be displayed:

```
0419 20 51 C1    b  jsr $C151
```

The lower-case **b** next to the instructions indicates that breakpoint is set at this location.

Next, enter the **go** command, which will begin running the application at full-speed. The screen will momentarily flash to the application screen and then back to the debugger screen. The following will be printed:

```
*** Software Breakpoint ***
0419 20 51 C1    b> jsr $C151
```

The **go** command began executing code beginning at the current location of the program counter, which was $400. Notice that the program counter is now at $419. This means that the instruction **jsr $C151** has not yet been executed, but is next on the list. However, all the code from $400 (**ProgStart** in the source code) to $419 (**jsr DoMenu** in the source code) was executed.

Watching the Menus Go Up

To view the current state of the application's screen display, press F7 . You will see a blank screen with the sprite pointer in the upper left corner. Press F7 again (or any other key) to return to the debugger screen. Notice that when you return to the debugger screen, all but the last line displayed is gone. Anytime you switch screens in the mini-debugger, you will lose all but the most recent line.

Now enter the **t** (top-step) command. The top step will single-step through the current instruction and return control to the command prompt at the next instruction. In the case of **jsr $C151**, the top-step will execute the subroutine **DoMenu** at full-speed and return when the next instruction (**lda #$04**) is encountered. If we wanted to, we could have single-stepped through the **DoMenu** subroutine using the **s** (single-step) command.

When the top-step returns, the next instruction, at the new location of the program counter, will be printed:

```
041C A9 04      > lda #$04
```

Now if you press [F7] to view the application's GEOS screen, you will see the effects of the **DoMenu** subroutine: the menus have been drawn. Return to the debugger screen by pressing any other key.

We can now top-step through the next two instructions by pressing [.] twice. If [.] is typed as the first character on a line, the previous command (in this case **t**) will be executed again. Pressing it twice should yield the following display:

```
041E 85 03      > sta $03
0420 A9 85      > lda #$85
```

The first press executed the **lda #$04** and displayed the next instruction to be executed (**sta $03**), and the second press executed the **sta $03** and displayed the next instruction to be executed (**lda #$85**).

Showing Registers

To view the current state of the processor's registers, enter the **r** (show registers) command:

```
Acc  X   Y    PC   SP NV-BDIZC  MemMap
$04 $00 $07 $0420 $FD 00100001 00110000
```

The two registers of interest here are the **Acc** (accumulator) and **PC** (program counter). The accumulator contains a $04, which is still around from the **lda #$04** at $41c, and the program counter is at $420.

Watching Register Values Change

Since the next instruction to execute is a **lda #$85**, an **s** single-step command ought to single-step through the instruction, thereby loading a $85 into the accumulator. Enter the **s** (single-step) command. You should see the following:

```
0422 85 02      > sta $02
```

The single-step executes the **lda #$85**, and then disassembles the next

instruction at $422. Now, enter the **r** (show-registers) command again:

```
Acc  X   Y    PC   SP NV-BDIZC  MemMap
$85 $00 $07 $0422 $FD 00100001 00110000
```

The register display shows that the accumulator (**Acc**) register now contains a $85, the result of the **lda**.

Running the Code Full-speed

Now that we've examined the application from within the debugger, let's run it full-speed. Use the **go** command. The **go** command displays the application's screen and begins execution at the current value of the program counter.

You can now experiment with the sample application. Click on the icon or select menu items. The application is running full-speed and has no idea that the debugger is lurking in the background just waiting to be called up in case of an error. Be careful, though, do not select **Quit** from the **file** menu because the application will attempt to leave to the deskTop and will be stopped by geoDebugger.

Hot Key Entry Into geoDebugger

When you are done playing with the application, press RESTORE. This is the geoDebugger "hot key." Whenever you press it, geoDebugger will immediately take control. The applications screen will be replaced with the debugger screen and the following message will be printed:

```
*** Execution stopped ***
FDAB 10 F7       > bpl $FDA4
```

The current location of the program counter (the instruction which was about to be executed when you pressed RESTORE) will be disassembled. The actual address will most likely be different than above because it depends on what instruction was being executed when you pressed RESTORE .

Whenever you enter geoDebugger with the RESTORE key, there is always a chance that the program will be in the middle of interrupt code. This is not problematic in itself, but can wreak havoc with disk I/O and some GEOS applications. Unless you are sure of what you are doing, it is always a good idea to execute a **sm** (stopmain) command. Do this now. **sm** sets a breakpoint in a safe place in GEOS **MainLoop** and then returns

to the program. Since most properly written GEOS applications will eventually return to **MainLoop**, the breakpoint will usually be encountered.

Modifying Program Data

One of the great benefits of a debugger is the ability to quickly modify an application and test the results. In geoDebugger it is very easy to modify instructions and program data. Say, for example, we don't like the way our menus look and we would like to modify them. Looking at the source code, we see that the text for the **geos** menu is stored at **GeosText**. But since the mini-debugger does not give us access to symbols, we have to find this text ourselves. Enter **d 400**. The **d** command means "dump memory" and shows us 128 bytes of data (in this case, starting at $400) in both binary and ASCII form:

```
        +0 +1 +2 +3 +4 +5 +6 +7 01234567
$0400 A9 C0 85 2F A9 04 85 03 )@./)...
$0408 A9 28 85 02 20 36 C1 A9 )(.. 6A)
$0410 04 85 03 A9 33 85 02 A9 ...)3..)
$0418 00 20 51 C1 A9 04 85 03 . QA)...
$0420 A9 85 85 02 20 5A C1 60 )... ZA
$0428 05 02 01 00 00 00 03 3F .......?
$0430 01 C7 00 00 0E 00 00 50 .G.....P
$0438 00 02 61 04 80 44 04 66 ..a..D.f
$0440 04 80 50 04 0F 1E 00 00 ..P.....
$0448 31 00 81 6B 04 00 10 05 1..k....
$0450 0F 2C 1D 00 40 00 82 7A .,..@..z
$0458 04 00 14 05 80 04 00 18 ........
$0460 05 67 65 6F 73 00 66 69 .geos.fi
$0468 6C 65 00 53 61 6D 70 6C le.Sampl
$0470 65 53 65 71 20 69 6E 66 eSeq inf
$0478 6F 00 63 66 6F 73 65 00 o.close.
```

Notice the geos text beginning at $461. This is the data for the menu entry. We want to modify this data, so we will use the **m** (memory) open command. The **m** command opens memory for display and modification as data. Enter **m 461** to begin modifying with the **g** in **geos** at $461. You will see the following:

```
0461 67          * .byte $67
                 ↑
              Cursor
```

Because the **m** command is an open mode, all further keystrokes, until the open mode is exited, will be interpreted by the **m** command. This allows the data display and entry to be interactive.

The $67 is the first character of the **geos** text string for the menu. the remaining characters (**e, o,** and **s**) are in the next three memory locations.

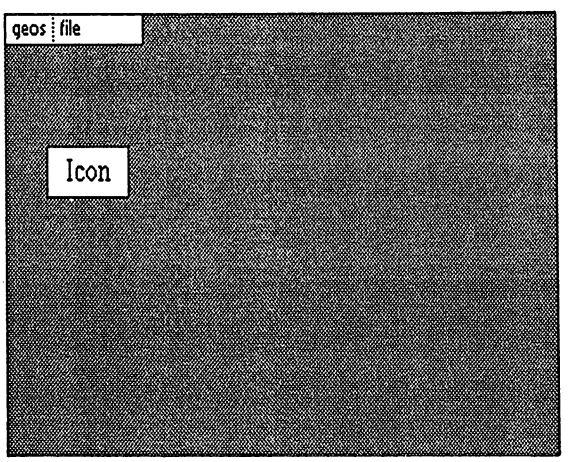
Press [SPACE]. The `$67` will disappear and the cursor will be moved into the data field of the **.byte**. Type **"GEOS"** (including the surrounding double-quotes) and press [RETURN]. Since the text string is four characters (four bytes) long, it will be deposited across four bytes, overwriting the lowercase **geos**:

```
0461 47          .byte $47
0462 45          .byte $45
0463 4F          .byte $4F
0464 53          .byte $53
```

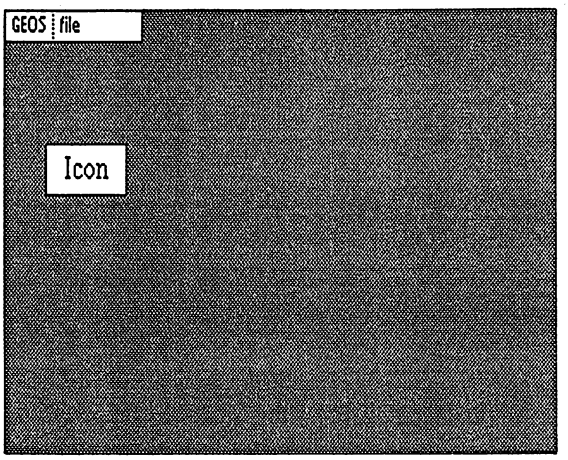
The data has now been modified in memory and control returned to the command prompt. (The **$47, $45, $4F, $53** are the hex equivalent of the ASCII codes for **GEOS**.)

Testing the New Menu

To run the program and see the new menu text, enter the **go** command:



But notice that the menu text hasn't actually changed. This is because, although the text data in memory has been modifed, the menu needs to be redrawn by GEOS before this change will be reflected in the display. To force a redraw of the menu, select an item from under the **geos** menu:



As soon as the menu is redrawn, the new uppercase **GEOS** menu will appear.

And Now on to More Powerful Manipulations

Now that you have completed the sample session with the mini-debugger, you should be beginning to feel comfortable with the debugging environment. You will find a complete discussion of the mini-debugger command set in chapter 9. If you begin to exhaust the limits of the mini-debugger by developing large applications, you should consider investing in a RAM-expansion unit and moving up to the super-debugger environment. Because the super-debugger is a superset of the mini-debugger, you will have access to all the familiar mini-debugger commands as well as the more powerful super-debugger commands.

Chapter 8: Super-debugger Reference

If you have a RAM-expansion unit connected to your Commodore, geoDebugger will automatically configure itself as a super-debugger. If you do not have a RAM-expansion unit, refer to the mini-debugger reference in Chapter 9.

This chapter contains a complete reference for the super-debugger configuration of geoDebugger. It covers every aspect of using the super-debugger, such as symbols, expressions, breakpoints, and macros, including a detailed description of each command. Although this is primarily a reference chapter, it would be a good idea to read it through completely at least once. For a general overview and information on using the super-debugger from the GEOS deskTop, refer to Chapter 7.

Special Characters

As in geoAssembler, some of the symbols and characters used by the super-debugger require special keystrokes. Additionally, because the standard text mode used by the super-debugger is unable to display the tilde (~) and the circumflex (^), they have been replaced with the pound (£) and arrow ( ) characters, respectively. To type any of these special characters in the super-debugger, use the following keystrokes:

Underline	_	C= + -
V-bar	|	C= + ↑
£-sign (replaces ~)	£	C= + * or £
up-arrow (replaces ^)		↑

Super-debugger Expressions

The super-debugger expression evaluator has some special features, symbols, and operators which are appropriate to the debugging environment, but it is otherwise identical to the expression evaluator in geoAssembler. For the most part, this chapter will only address the differences between the two evaluators. For more information on the geoAssembler expression evaluator, refer to "Expressions" in Chapter 5.

Numeric Constants

The super-debugger expression evaluator, like the geoAssembler expression evaluator, will work with decimal, hexadecimal, octal, or binary constants, as well as character constants. However, the super-debugger supports the option of changing the default radix (number base). In geoAssembler, the default radix is decimal, and it cannot be changed; to specify any other radix, the number must be preceded by a special symbol, such as $ for hexadecimal. the super-debugger, on the other hand, allows the default radix to be either decimal or hexadecimal. Because most debugging is done in hexadecimal, the super-debugger defaults to that radix, and any number which is not preceded by a radix symbol will be considered hexadecimal. Using the **opt** command, you can change the default radix to decimal, like it is in geoAssembler. In any case, regardless of the default radix, you can always precede a number by its appropriate radix symbol.

Decimal:	A period followed by a string of decimal digits (0-9). If decimal is the default input radix, the period is optional. Example: **.1234**
Hexadecimal:	A dollar sign ($) followed by a string of hexadecimal digits (0-9, a-f). If hexadecimal is the default input radix, the dollar sign is optional. Example: **$4f9c**
Octal:	A question mark (?) followed by a string of octal digits (0-7). Example: **?07117**
Binary:	A percent sign (%) followed by a string of binary digits (0,1). Example: **%11001010**

Character: A single ASCII character enclosed in single-quotes ('). The character is converted to a 16-bit value with the high-byte set to zero.
Example: **'A'**

NOTE: when the default radix is hexadecimal, there are a couple of idiosyncracies to be aware of. First, when assembling code a lone **A** or **a** as in **lsr a** will be interpreted as accumulator addressing mode as opposed to a **$a**; use the **$** radix symbol to avoid this confusion. Second, avoid defining symbols which look like hex values (e.g., **fed**, **aaa**, **abc**); they will be interpreted as hexadecimal values unless the default radix has been changed to decimal.

Symbol Names

Any symbol in the the super-debugger symbol table can be used within an expression. When the expression is evaluated, the symbol is replaced with its absolute value. Symbols can be entered into the symbol table by either loading a .dbg symbol file (created by geoLinker) or by entering the symbol manually (e.g., with the **sym** command).

Unlike geoAssembler, symbols in the super-debugger may be referenced without case distinction. That is, **mouseon** (all lower-case) can be used to refer to **mouseOn** or **MOUSEon** (mixtures of upper- and lower-case); the case will not be significant. Experience has shown that most symbols are unique without case distinction, and ignoring the case makes them easier to enter and manipulate in the debugger. If you have symbols which depend on case distinction, you can always enable case checking with **opt** command.

Processor Registers

The super-debugger expression evaluator also gives you access to the current values of the six processor registers and the Commodore memory map register. Registers are referenced with an **r**, a period (.), and a letter code for the register:

r.a	accumulator	(one byte)
r.x	X-index	(one byte)
r.y	Y-index	(one byte)
r.st	processor status	(one byte)
r.pc	program counter	(two bytes)
r.sp	stack pointer	(two bytes)
	(Note: although the stack pointer is actually a one-byte index into page one, the number returned with ***r.sp*** *is a two-byte address which actually points to the top of the stack.)*	
r.mm	Commodore memory map	(one byte)

The only non-standard register is the Commodore memory map. This value is not a true 6502 register, but it is of similar importance. It is picked up from location $0001 of the Commodore memory space and indicates the current state of the switchable memory banks. For information on interpreting this value, refer to the *Commodore 64 Programmer's Reference Guide*.

Status Register Flags

In addition to giving access to the full byte value of the processor status register (with **r.st**), the expression evaluator lets you access the individual flags (bits) within that register. Status register flags are referenced with an **f**, a period (.), and a letter code for the flag:

f.n	negative flag
f.v	overflow flag
f.b	break flag
f.d	decimal mode flag
f.i	interrupt disable flag
f.z	zero flag
f.c	carry flag

All flags are either one (true) or zero (false).

User and System Variables

There are ten user variables and four system variables which are accessible in expressions. These variables are referenced with a **u**, a period (.), and a code for the variable:

User Variables

u.0	user variable 0
u.1	user variable 1
u.2	user variable 2
u.3	user variable 3
u.4	user variable 4
u.5	user variable 5
u.6	user variable 6
u.7	user variable 7
u.8	user variable 8
u.9	user variable 9

System Variables

u.lc	location counter: returns the address of the most recently opened memory location.
u.ws	window size: the total number of printable screen lines.
u.wc	window counter: the total number of lines printed since the last user-input; this can be used in conjunction with **u.ws** to detect when the screen is full.
u.fn	current value of **for** macro loop counter.

IMPORTANT: Changing the value of **u.ws** (window size) can lead to unpredictable results. Currently this value is a constant 24, but in future implementations this may change.

Operators

The operators in the super-debugger expression evaluator are identical to those in geoAssembler, except for three new operators and new representations for two other operators. (The representations of the geoAssembler ~ and ^ operators have changed in the super-debugger because the tilde and circumflex characters cannot be displayed in Commodore text mode.) The following table shows all of the valid operators and their precedence. Operators with a ‡ in the left margin only exist in the super-debugger; operators with a † symbol in the left margin exist in both the super-debugger and geoAssembler but have different character representations.

	OPERATOR	PRECEDENCE	
	( )	1	grouping parentheses (sub-expression)
	-	2	unary negation
	!	2	logical not
†	£ (~)	2	bitwise one's complement
	[or <	2	low-byte
	] or >	2	high-byte
‡	@	2	byte lookup at address
‡	@@	2	low/high word lookup at address
‡	@#	2	length of 6502 instruction at address
	**	3	exponentiation
	*	4	multiplication
	/	4	division
	//	4	modulus
	+	5	addition
	-	5	subtraction
	>>	6	logical shift right
	<<	6	logical shift left
	>	7	logical greater than
	>=	7	logical greater than or equal to
	<	7	logical less than
	<=	7	logical less than or equal to
	= = or =	8	logical equal
	!=	8	logical not equal
	&	9	bitwise and
†	↑ (^)	10	bitwise exclusive-or (xor)
	|	11	bitwise inclusive-or (ora)
	&&	12	logical and
†	↑↑ (^^)	13	logical exclusive-or
	||	14	logical inclusive-or

Operator: @

Byte lookup at address. This unary operator looks into the application's memory space and returns the byte at the address represented by its argument.

Examples:

@$3000 returns the byte value stored at address $3000

@my_sym assuming **my_sym** is defined in the symbol table, returns the byte value pointed to by **my_sym**.

@(r.sp+1) the top byte on the stack. (Remember: the SP points to the next *available* byte, not the byte just pushed; we add one to compensate.)

Operator: @@

Low/high word lookup. This unary operator looks into the application's memory space and returns the low/high word at the address represented by its argument.

Examples:

@@$3000 returns the word value stored at address $3000 and $3001; the byte at $3000 is used as the low-byte and the byte at $3001 is used as the high-byte.

@@my_sym assuming **my_sym** is defined in the symbol table, returns the word value stored at address **my_sym** and **my_sym+1**; the byte at **my_sym** is used as the low-byte and the byte at **my_sym+1** is used as the high-byte.

@(@@jump) assuming **jump** is defined in the symbol table, returns the byte pointed at indirectly by the low, high address stored at **jump**.

Operator: @#

Instruction length calculation. This unary operator looks into the application's memory space and returns the length (in bytes) of the 6502 instruction located at that address. If the address contains an invalid opcode, a zero ($0000) will be returned.

Examples:

@#$3000 returns the length of the 6502 instruction which begins at $3000.

@#my_prg assuming **my_prg** is defined in the symbol table, returns the length of the 6502 instruction which begins at address **my_prg**.

The remaining operators are identical to those found in geoAssembler. Refer to "Operators" in Chapter 5 for more information.

Basic Operation

The Command Prompt

The basic geoProgrammer command prompt is a greater-than (>) symbol in the leftmost column at the bottom of the screen. Whenever this prompt is displayed, geoProgrammer is idle, awaiting a command. You can type commands in at this point. The following keystrokes have an effect in this mode:

RETURN	Enters the current line; the super-debugger will attempt to interpret and process the command.
DEL	Deletes the character to the left of the cursor.
←	Erases the current input line.
,	Reprints the last command on the current input line, which allows the command to be edited and then re-entered with RETURN . The comma must be typed as the first character on the input line.
.	Repeats the last command. This is similar to pressing , followed by RETURN . The period must be typed as the first character on the input line.

Hot Key Entry and Cancel

When your program is running, the RESTORE key acts as a "hot key"; it will suspend execution and enter the debugger. When you are in the super-debugger, RESTORE will cancel a command or a macro and return to the input prompt at any time. Because of a hardware limitiation in the Commodore keyboard, you may have to press RESTORE a couple of times to get it to respond.

The More Prompt

The screen print routine monitors the **u.ws** (window size) and **u.wc** (window count) system variables. Each time it prints a line without returning to the command prompt, the count variable is incremented. When the count variable exceeds the window size, the screen is full of text; the print routine will detect this and pause, displaying a "more" prompt and

awaiting input before it will continue. At the prompt you can press the space bar to get another full screen of text or you can press RETURN to get just one more line.

SPACE	full screen of text.
RETURN	one more line.

Viewing the GEOS Application Screen

You can switch between the super-debugger text screen and the GEOS application's hi-res screen at any time by the pressing the F7 key. You can return to the debugger screen by pressing any other key.

EnterDeskTop Vector Trap

geoDebugger sets an permanent breakpoint at the GEOS **EnterDeskTop** vector. If an application attempts to exit by calling **EnterDeskTop**, the following will be printed:

```
*** EnterDeskTop vector encountered ***

C22C EnterDes     >brk
```

When geoDebugger is running, an application cannot be allowed to leave to the deskTop directly. geoDebugger must first remove itself in order for the deskTop to function properly. To return to the deskTop, use the super-debugger **quit** command.

Super-debugger Command Summary

General Commands

quit	Exits geoDebugger and returns to the deskTop.
opt	Super-debugger configuration options.

General Display Commands

r	Display processor registers.
dump	Display a block of memory in hex and ASCII format.
n	Disassemble code nearby (above and below) the program counter.

w	Disassemble a window of code from program counter down.
dis	Disassemble a full screen of code.
print	General value, symbol, and expression print.

Open Modes (register and memory examination and modification)

a	Open memory as assembly language code.
m	Open memory as data.
reg	Open processor registers.
flag	Open processor status register as individual flags.

Execution Commands

go	Start full speed execution of program.
runto	Set breakpoint and go.
jsr	Execute subroutine at address.
s	Single-step through current level and subroutines.
t	Single-step through current level and top-step through subroutines.
p	Proceed with execution at full-speed until breakpoint.
next	Proceed until next instruction is reached (for exiting loops).
loop	Proceed until a full loop is completed.
skip	Skip over the current instruction without executing it.
stopmain	Stopmain; stop execution in GEOS **MainLoop**.

Stack Related Commands

stack	Display the top eight bytes on the stack.
history	Display current step-through-jsr history.
inithist	Initialize current step-through-jsr history.
finish	Finish up most recent subroutine that was single-stepped into.
return	Run until subroutine returns.

Breakpoint Commands

b	Display breakpoints.
setb	Set a breakpoint.
clrb	Clear a breakpoint.
initb	Initialize breakpoint table, clearing all breakpoints.

Symbol Commands

sym	Display symbols.
setsym	Define a symbol.
clrsym	Clear a symbol.
initsym	Initialize (clear) symbols from currently active modules.
mod	display symbol priority of overlay modules.
setmod	Set symbol priority of overlay modules.
initmod	Initialize overlay module priority tables.

Macro Commands

sysmac	Display system macros.
mac	Display user-defined macros.
setmac	Define user macro.
clrmac	Clear user-defined macro.
initmac	Initialize (clear) all user-defined macros.
poff	Printing off.
pon	Printing on.
if	Conditional.
for	Loop.
stop	Stop macro execution and return to command prompt.

Memory Commands

find	Find a pattern in memory. 8-95
fill	Fill memory with a pattern.
copy	copy a block of memory.
diff	compare two blocks of memory.

Special Commands

setu	Set user variable.
pc	View and set program counter.
rboot	Reboot GEOS.

Disk Commands

drivea	Make drive A the current drive.
driveb	Make drive B the current drive.
disk	Display name of disk in current drive.
dir	Display directory of disk in current drive.
getb	Get disk block from current drive.
putb	Put disk block to current drive.
getn	Get next logical block from current drive.
getchain	Get logical chain of blocks from current drive.
dumpd	Display disk buffer in hex and ASCII format.

Syntax Notation

The following conventions are used in the syntax descriptions of the super-debugger commands. Much of this notation will be familiar from geoAssembler and geoLinker.

exp — a valid expression.

string. — a string of ASCII characters enclosed in double-quotes.

symbol — a valid geoAssembler type symbol name.

macname — a macro or system macro (command) name.

range — describes a range of values in one of the following forms:

- *exp* — a single value (a range of one).
- *exp*:*exp* — a start/finish range (ranges from the first expression to the second, with lowest value first).

  Example: **$1000:$2000** *ranges from $1000 to $2000.*
- *exp*:#*exp* — a start/count range (ranges from the first expression for a count expressed in the second expression).

Example: **Buffer:#.200** *ranges from the address of* ***Buffer*** *to* ***Buffer+.200.***

searchspec describes a search specification for label and macro names. A *searchspec* is made up of valid symbol characters (letters, numbers, and the underscore symbol) and the **?** and * wildcards. A **?** anywhere in the searchspec will match a single character, and a * will match any number of characters.

Example: **symb*** *would match with* ***symbol1***, ***symbol2***, ***symb_3er***, *and* ***symbat.***

Example: **??mbo*** *would match with* ***symbol1***, ***symbol2***, ***t3mbol_i***, *and* ***rambo86.***

Example: **????** *would match with all names with exactly four characters.*

breakcond describes a conditional breakpoint specification in one of the following forms:

exp a counter. Each time the breakpoint is encountered, the counter is decremented; when it goes to zero, the break succeeds.

Example: **5** *will pass through the breakpoint four times; the break will succeed on the fifth time through.*

=exp a condition. Each time the breakpoint is encountered, the expression is evaluated; the break will only succeed when the expression evaluates to true.

Example: **=(r.x > 30)** *will pass through the breakpoint until the X-register exceeds thirty.*

exp,=exp combination counter and condition. Each time the breakpoint is encountered, the expression is evaluated. If the condition is true, the counter is decremented. When the counter goes to zero, the break succeeds.

Example: **3,=(f.c && @cmd==4)** *will pass through the breakpoint waiting for the carry flag to be set and the variable* **cmd** *to be equal to four; when this happens three times, the break will succeed.*

[ ] square brackets indicate an optional item which may appear zero or one times.

{ } curly braces indicate an optional item which may appear one or more times.

| a vertical line indicates a choice and can be read as "or"

In addition, all sample output from the super-debugger will be printed in a `bold courier` font so that the spacing will closely match the standard Commodore text mode.

General Commands

Command: **quit**

Synonymn: **exit, q, e**

Mini: see **q** in Chapter 9.

Purpose: leave the super-debugger and return to the GEOS deskTop.

Usage: **quit**

Note: takes no parameters.

quit leaves the super-debugger and returns to the GEOS deskTop by disabling itself and performing a standard application exit (calls **EnterDeskTop**). The program space will be cleared and all debugger symbols and macros will be lost. If GEOS was corrupted during the debugging session (trampling the memory from $c000 to $a000 is a great way to do this), **quit** will very likely crash the system, leaving you no alternative but to reboot by turning off the power. In instances where you fear GEOS has been destroyed, the **rboot** command should be used for leaving the super-debugger.

Before actually leaving, you will be asked you confirm your intention to quit:

```
Exit to deskTop (y/n)?
```

Typing [Y] will exit; typing [N] or any other key will return to the command prompt.

Command: **opt**

Mini: see **g0** and **g1** in Chapter 9.

Purpose: set super-debugger options.

Usage: **opt** [*optnum*] | [*optnum,setting*]

Note: *optnum* is an expression which evaluates to a valid option number (0-6), and *setting* is an expression which evaluates to an appropriate setting number for that option (0 or 1).

There are seven super-debugger configuration options:

Option		Settings	
0	input radix	0	hexadecimal (default)
		1	decimal
1	output radix	0	hexadecimal (default)
		1	decimal
2	labels	0	enabled (default)
		1	disabled
3	offset radix	0	hexadecimal (default)
		1	decimal
4	case distinction	0	disabled (default)
		1	enabled
5	GEOS screen	0	disabled (default)
		1	enabled
6	expand macros	0	disabled (default)
		1	enabled

Input radix (0). The input radix is the default number base used within expressions. If this is set to hexadecimal, the **$** symbol is optional in front of hexadecimal numbers; if this is set to decimal, the **.** (period) symbol is optional in front of decimal numbers.

Output radix (1). The output radix is the default number base used for output from the **print** command, the **m** open command, and data appearing in disassembled output, as with the **a** command. The appropriate radix symbol (**$** or **.**) will always be printed along with the number.

Labels (2). When labels are enabled, disassembly and data viewing commands, as with the **a** and **m** commands, will display the label plus

offset for the absolute address of code and memory. When labels are disabled, the hex byte values at the location will be displayed. See also: **a** and **m**.

Offset radix (3). Numbers printed as offsets from symbols appear as `symbol+xxx`, where *symbol* is the symbol name and *xxx* is a one byte offset. The offset can be shown in either hexadecimal or decimal. If the offset radix is hexadecimal, a $ radix symbol will precede the number; if the offset radix is decimal, no radix symbol will be printed.

Case distinction (4). If case distinction is disabled, symbols may be typed in expressions without regard to the actual upper- and lower-case name as defined in geoAssembler. If case distinction is enabled, the upper- and lower-case must match the symbol exactly. For more information, refer to "Symbol Names" in this chapter.

GEOS screen (5). If GEOS screen is enabled, while processing a command which executes code, such as **s**, **t**, or **next**, the super-debugger will display the application's screen. If GEOS screen is disabled, the application's screen will only be shown during a **go**, **runto**, or when F7 is pressed.

Expand macros (6). If macro expansion is enabled, then the macro stream will be echoed to the debugger screen. This gives you a visual audit-trail of a macro's activities.

opt in Open Mode

The most straightforward way changing options is to enter **opt** without any parameters. the super-debugger will open the last option opened and allow you to change the value interactively. When **opt** is in open mode, the display will appear as:

option number	option description	current option selected
opt0	Input radix:	* (0) hexadecimal

When an option is opened, the **opt** command is intercepting keystrokes and responding at that level. There are four keystrokes which have an effect in this mode:

SPACE	toggle current option setting.
0	set option to setting 0.
1	set option to setting 1.
SHIFT + ↑↓	close current option and open previous option.
↑↓	close current option and open next option.
RETURN	close current option and return to command prompt.

With the ↑↓ key you scroll through the options, changing them with SPACE as you please.

To open a specific option, enter **opt** followed by an option number (0-6). For example,

opt 4

Would open option four (case distinction). All the same open mode keys are active.

Using opt Without Open Mode

You can change an option without actually opening it by providing a setting along with the option number when entering the command. For example,

opt 0,1

will set the output radix to decimal (setting 1).

Display Commands

Command: **r**

Mini: See **r** in Chapter 9.

Purpose: display processor registers.

Usage: **r**

Note: takes no parameters.

The **r** command displays all the processor registers, including the MM (memory map) pseudo-register. The output is in the following format:

```
Acc  X   Y    PC   SP NV-BDIZC  MemMap
$00 $00 $00 $0400 $FF 10000011 00110000
```

The accumulator (Acc), x-register (X), y-register (Y), and stack pointer (SP) are all printed as one-byte hexadecimal values. The program counter (PC) is a two-byte hex value. The processor status register is a one-byte binary value. The **NV-BDIZC** notation above the bits refers to the individual flags in the status register; a one means the flag is set, a zero means it is clear. The memory map register is printed as a one-byte binary value.

For more information on the processor registers, refer to "Processor Registers" in this chapter and a book on 6502 assembly language.

See also: **reg** and **pc**.

Command: **dump**

Synonymns: **d**

Mini: See **d** in Chapter 9.

Purpose: dumps 128 ($80) bytes to the screen in hexadecimal and ASCII.

Usage: **dump [*exp*]**

Note: *exp* is the starting address for the dump. If no address is specified, the value of the current location counter (**u.lc**) will be used.

dump is used to view 128 bytes of memory at once. It fills almost the entire screen with information and is especially useful for looking at tables and buffers. The super-debugger will dump memory from the nearest eight-byte boundary which includes the specified address. 128 bytes are dumped, eight bytes per screen line. The address of the first byte in each line is printed at the left margin, followed by the eight bytes of data (corresponding to the +0 to +7 offsets), followed by eight ASCII characters. Note: if a character cannot be printed on the screen, it will be displayed as a period.

Example:

dump $3080 might produce the following display:

```
     +0 +1 +2 +3 +4 +5 +6 +7 01234567
$3080 73 B1 88 03 13 20 71 A4 s1... q$
$3088 54 48 4B 2C 20 4D 47 4C THK, MGL
$3090 2C 20 61 6E 64 20 45 44 , and ED
$3098 53 20 77 65 72 65 20 68 S were h
$30A0 65 72 65 2E D0 18 00 2C ere.P..,
$30A8 F0 F0 18 00 2C 18 00 8D .p......
$30B0 04 A9 AA 0F 29 8A 70 85 .)*.).p.
$30B8 4A A5 78 08 60 18 00 8E J%x. ...
$30C0 02 A2 00 F0 C1 D0 18 00 ."pAP..
$30C8 41 53 43 49 49 2A EA 18 ASCII*J.
$30D0 54 45 58 54 BD 70 A6 18 TEXT=p&.
$30D8 00 A9 F9 8F 20 33 84 18 .)y. 3..
$30E0 00 8C 1C 00 8D F7 29 1C .....w).
$30E8 00 AD 04 89 20 F0 F0 18 .-.. .p.
$30F0 00 2C 04 A9 04 57 20 48 .,.).W H
$30F8 49 FF B7 B7 FF B7 B7 BF I.77.77?
```

Command: **n**

Purpose: disassemble code in the neighborhood of the current program counter. Displays five lines of code: two before the program counter, followed by three more, including the program counter.

Usage: **n**

Note: takes no parameters.

The **n** command disassembles the code surrounding the current program counter. It is useful for seeing where a program is going as well as where it came from. The output is in the standard disassembly format as described under the **a** command.

Example:

With the program counter at $0404, an **n** might produce the following output:

hex address	*label plus offset*	*disassembly*
0400	ProgStar	lda #$C0
0402	ProgStar+$02	sta dispBuff
0404	ProgStar+$04 >	lda #$04
0406	ProgStar+$06	sta r0h
0408	ProgStar+$08	lda #$28

NOTE: Because the **n** command must backtrack to show instructions in front of the program counter, it may not have enough information from context to correctly synchronize with the instruction boundaries. the super-debugger has a fairly sophisticated algorithm for synchronizing and will almost always do so successfully when there is legitimate code before and after the program counter.

See also: **w, dis, pc.**

Command: **w**

Mini: See **w** in Chapter 9.

Purpose: disassembles a window of code at the program counter. Displays five lines of code, starting with the current program counter location.

Usage: **w**

Note: takes no parameters.

The **w** command disassembles five lines of code beginning with the current program counter. It useful for seeing the instructions about to be executed. The output is in the standard disassembly format as described under the **a** command.

Example:

With the program counter at $0404, a **w** might produce the following output:

```
hex
address  label plus offset        disassembly

0404     ProgStar+$04 >     lda #$04
0406     ProgStar+$06       sta r0h
0408     ProgStar+$08       lda #$28
040A     ProgStar+$0A       sta r0l
040C     ProgStar+$0C       jsr Graphics
```

See also: **n, dis, pc.**

Command: **dis**

Purpose: disassembles a full screen of code.

Usage: **dis** [*addrexp*]

Note: *addrexp* is the starting address for the disassembly. If no address is specified, the value of the current location counter (**u.lc**) will be used.

The **dis** command disassembles a full screen of code. The output is in the standard disassembly format as described under the **a** command.

Example:

Assuming **ProgStar** is a label defined in the symbol table as $400, a **dis ProgStar** might produce the following output:

```
hex
address   label plus offset      disassembly

0400      ProgStar              lda #$C0
0402      ProgStar+$02          sta dispBuff
0404      ProgStar+$04 >        lda #$04
0406      ProgStar+$06          sta r0h
0408      ProgStar+$08          lda #$28
040A      ProgStar+$0A          sta r0l
040C      ProgStar+$0C          jsr Graphics
040F      ProgStar+$0F          lda #$04
0411      ProgStar+$11          sta r0h
0413      ProgStar+$13          lda #$33
0415      ProgStar+$15          sta r0l
0417      ProgStar+$17          lda #$00
0419      ProgStar+$19          jsr DoMenu
041C      ProgStar+$1C          lda #$04
041E      ProgStar+$1e          sta r0h
0420      ProgStar+$20          lda #$85
0422      ProgStar+$22          sta r0l
0424      ProgStar+$24          jsr DoIcons
0427      ProgStar+$27          rts
0428      ClearScr              ora r0l
```

Note: The **dis** command sets the location counter (**u.lc**) to point at the instruction following the last instruction disassembled. This way a subsequent **dis** (without a parameter) will continue the disassembly.

See also: **n, w, pc.**

Command: **print**

Synonymn: **pr**

Purpose: general purpose expression, string, and symbol printing.

Usage: **print** ***printitem*{,*printitem*}**

Note: *printitem* is a complex construction which is described below. You may supply up to ten *printitems.*

The **print** command is a powerful and flexible output facility. At its simplest, it is useful for evaluating symbols, expressions, and doing number-base conversions. At its full sophistication, it can be used for complex formatted output, such as with the **dir** command.

Brief Introduction to Using print

Because the **print** command is so sophisticated, a few of its most useful features will be introduced.

To print a string to the screen (useful in a macro), simply enclose the text in quotes:

print "This string will be printed."

To print the result of an expression in the default radix, use the expression as the only parameter:

print ($1000+symbase) | $8000

print .65536/.16

To print the result of an expression in any radix, follow the expression with a colon and the radix symbol:

print ($1000+symbase) |$8000:% *print result in binary (%).*

print .65536/16:s *print as symbol.*

These are the rudiments of the **print** command. A full description of the command follows.

print Syntax

The parameter format for the **print** command adheres to the following syntax:

printitem — *string* | *printexp*

printexp — *exp*[:*printoption*]

printoption — [*namestring*][*lookup*][*radix*]
an optional formatting code.

namestring — *string*
a string which will be placed before the output, replacing the standard echo of the expression.

lookup — [*decnum*]*datasize*
For doing byte or word lookups at the address of the expression.

decnum — a decimal number without a radix sign; used in conjunction with *datasize* to indicate the number of byte or word lookups to perform at the address of the expression. If *decnum* is omitted, one byte or one word will be looked up.

datasize — **b|w**
a specifier indicating the size of the data to be looked up: either byte or word. The number of bytes or words as determined by *decnum* will be looked up beginning at the address of the expression and output to the screen.

radix .|$|?|%|s|'

Indicates how the output should be printed:

. decimal
$ hexadecimal
? octal
% binary
' character
s symbolic+offset (if possible)

Examples:

print $5f4a:. *converts **$5f4a** to decimal and prints the result.*

print .1000+?20:s *prints out the result of **.1000** plus **?20** in symbolic form.*

print "registers:",r.a,r.x,r.y
prints a string followed by the contents of the accumulator and the x- and y-registers.

print u.lc:16b$ *prints out 16 bytes in hexadecimal, starting at the address currently in the location counter.*

print ibuffer:30b'"Text input buffer: "
*prints 30 characters starting at the address of the symbol **ibuffer** and names the output **Text input buffer.***

Open Modes

Command: **a**

Mini: see **a** in Chapter 9.

Purpose: open memory for assembly language code

Usage: **a** [*addrexp*]

Note: *addrexp* is the memory address to open. If no parameter is specified, the current address pointed to by the location counter (**u.lc**) will be opened.

The **a** command is the general disassemble, assemble, and modify open command. When you open a memory location with **a**, you are placed in an interactive mode where all keystrokes are intercepted and handled specially. In **a**-mode you are able to disassemble code forward and backward, define labels, and modify instructions at any point.

Output for the **a** command is in the following general format, although certain fields may be displayed differently if you have changed the default options with the **opt** command:

```
hex
address  label plus offset      flag  disassembly

0400   ProgStar                      lda #$C0
0402   ProgStar+$02                 sta dispBuff
0404   ProgStar+$04          >      lda #$04
0406   ProgStar+$06                 sta r0h
0408   ProgStar+$08          b      lda #$28
040A   ProgStar+$0A            *    sta r0l
```

hex address is the absolute address of the instruction. Instructions are either one, two, or three bytes in length.

label plus offset is either a label with a positive one byte ($00-$ff) offset or the absolute address if there is no label within $ff bytes backward. If you disable labels with the **opt** command (option 2) or you toggle the display with the C open-mode keystroke, this field will contain the hexadecimal bytes which comprise the instruction, as in the following example:

```
0408  A9 28              lda #$28
```

where **A9** is the hexadecimal value for **lda** immediate, and **28** is the hexadecimal value for **#$28**.

flag is a field with three positions, each of which has a unique possible symbol:

b	breakpoint set at this instruction.
>	program couter points at this instruction.
*	current opened instruction.

disassembly is a disassembly of the bytes at the address. If the location does not contain a valid 6502 opcode, `???` will be displayed.

Open a-mode Keystrokes

When memory is opened with the **a** command, the super-debugger is intercepting keystrokes and responding at that level. When an invalid keystroke or a bad entry is detected, the cursor will briefly flash as a **?** symbol. When the cursor is on the asterisk in the *flag* field, the following keystrokes will have an effect:

SPACE	enter deposit mode at this location (see deposit description below)
↓ or J	close current instruction and open next instruction.
SHIFT + ↓ or K	close current instruction and open previous instruction.
←	reopen current instruction.
RETURN	close current instruction and return to command prompt.

M	switch from **a**-mode to **m**-mode. See: **m** command.
•	display as decimal.
$	display as hexadecimal.
?	dispaly as octal.
%	display as binary. Note: word values will be displayed as hexadecimal because low/high binary words are seldomly useful.
!	display as characters.
S	display in symbolic form.
C	change label enable/disable status (refer to option 2 under **opt**).
B	set breakpoint at this address.
>	set program counter to this address.
L ↕	open at address of next label.
L SHIFT + ↕	open at address of previous label.
L −	delete currently displayed label, even if it is displayed with an offset. If the same label exists in multiple modules, the label will only be deleted from the module with the highest priority.

Deposit a-mode

When you press SPACE at the asterisk prompt, the disassembly field clears and the cursor is placed into it. At this point you can enter a new 6502 instruction. As on the command line, DEL deletes the character to the left of the cursor and ← clears the input line.

To enter a line and leave deposit mode, use one of the following keystrokes:

←	enter current line and reopen current instruction. (Useful for checking a complex operand expression or entering symbol and then an instruction.)
↓	enter current line and open next instruction.
SHIFT + ↓	enter current line and open previous instruction.
RETURN	enter current line and return to command prompt.

If an error is detected in the entry, the line will not be entered and the cursor will briefly flash as a ?.

To leave deposit mode without entering a line, do one of the following:

1. Enter an empty line or a line which contains only spaces.

2. Use DEL to backspace out of the disassembly/deposit field.

a-mode Deposit Syntax

The a-mode deposit entry must be a valid 6502 opcode/operand construction as in geoAssembler. Because the mini-debugger does not support expressions or any radix other than hexadecimal, any numbers in the operand must conform to this limitation. Also: you cannot type beyond the left edge of the screen. If you try this, the cursor will briefly flash as a ?.

Example deposit entries:

lda #$fe	*opcode and hexconst immediate value.*
sei	*opcode alone.*
jsr 33ef	*opcode and hexconst address.*

Command: **m**

Mini: see **m** in Chapter 9.

Purpose: open memory for data.

Usage: **m** [*addrexp*]

Note: *addrexp* is the address to open. If no parameter is specified, the current address pointed to by the location counter (**u.lc**) will be opened.

m is the general view and modify data command. When you open a memory location with **m**, you are placed in an interactive mode where all keystrokes are intercepted and handled specially. In **m**-mode you are able to view data forward and backward and modify it at any point.

Output for the **m** command is in the following general format:

```
hex
address label plus offset     flag  mode   data

046B 53                             .byte $53
046C 61                             .byte $61
046D 6D                             .byte $6D
046E AboutTex+$03                   .byte $70
046F AboutTex+$04                   .byte $6C
0470 AboutTex+$05  *                .byte $65
```

hex address is the absolute address of the data.

label plus offset is either a label with a positive one byte ($00-$ff) offset or the absolute address if there is no label within $ff bytes backward. If you disable labels with the **opt** command (option 2) or you toggle the display with the C open-mode keystroke, this field will contain the hexadecimal bytes which comprise the data, as in the following examples:

```
0470 65                 .byte $65
046D 6D 70              .word $706D
```

This feature is especially useful when you are displaying the data in a different radix — you will still have immediate access to a hexadecimal representation.

flag is a field with three positions, each of which has a unique possible symbol:

b	breakpoint set at this instruction.
>	program counter points at this instruction.
*	Current opened instruction.

mode is the data display mode, either `.byte` or `.word`. Data shown in word format is displayed in low/high order as in the following example:

```
046D AboutTex+$02  .word $706D
```

data is the actual data at the current address. The data will not undergo symbol substitution unless you request it specifically with the [S] key (see below).

Open m-mode Keystrokes

When data is opened with the **m** command, the super-debugger is intercepting keystrokes and responding at that level. When an invalid keystroke or a bad entry is detected, the cursor will briefly flash as a `?` symbol. When the cursor is on the asterisk in the *flag* field, the following keystrokes will have an effect:

[SPACE]	enter deposit mode at this location (see deposit description below)
[↓] or [J]	close current location and open next one.
[SHIFT] + [↓] or [K]	close current location and open previous one.
[↵]	reopen current location (useful for rereading a hardware location with changing values).
[RETURN]	close current location and return to command prompt.

D B	display data as byte.
D W	display data as word.
A	switch from **m**-mode to **a**-mode. See: **a** command.
.	display data as decimal.
$	display data as hexadecimal.
?	display data as octal.
%	display data as binary. Note: word values will be displayed in hexadecimal because low/high binary words are seldom useful.
'	display data as characters.
S	display data in symbolic form.
C	change label enable/disable status (refer to option 2 under **opt**).
B	set breakpoint at this address. (not extremely useful when looking at data.)
>	set program counter to this address. (not extremely useful when looking at data.)
L ↑↓	open at address of next label.
L SHIFT + ↑↓	open at address of previous label.
L -	delete currently displayed label, even if it is displayed with an offset. If the same label exists in multiple modules, the label will only be deleted from the module with the highest priority

Deposit m-mode

When you press space at the asterisk prompt, the data field clears and the cursor is placed into it. At this point you can enter new data for this address. As on the command line, DEL deletes the character to the left of the cursor and ← clears the input line.

To enter a line and leave deposit mode, use one of the following keystrokes:

←	enter current line and reopen current instruction. (Useful for checking a complex operand expression.)
↓	enter current line and open next instruction.
SHIFT + ↓	enter current line and open previous instruction.
RETURN	enter current line and return to command prompt.

If an error is detected in the entry, the line will not be entered and the cursor will briefly flash as a `?`.

To leave deposit mode without entering a line, do one of the following:

1. Enter an empty line or a line which contains only spaces.

2. Use DEL to backspace out of the disassembly/deposit field.

m-mode Deposit Syntax

m-mode deposits for `.byte` and `.word` deposits is slightly different:

`.byte` *string* | *exp*{,*exp*}

You cannot deposit more than 40 bytes (40 characters or 40 values) in a single deposit. Expressions *must* evaluate to a byte value ($00-$ff). If in doubt, use the [grab byte operator.

`.word` *exp*{,*exp*}

You cannot deposit more than 40 words in a single deposit.

The full deposit entry may be up to 100 characters in length. If you try to type beyond the 100 character limit, the cursor will briefly flash as a `?`.

Example deposit entries:

```
.byte "This is a string"

.byte $00,$ff,[prog_star-.37,'c','T'

.word $6543,AboutTex*2/4
```

Command: **reg**

Synonymn: **rg**

Mini: see **rg** in Chapter 9.

Purpose: open register.

Usage: **reg [[r.]*regname*]**

Note: *regname* is the optional register name to open (note a **r.** may be appended to the beginning of the register name):

a | x | y | sp | pc | st | mm

If no register is specified, the last register opened will be reopened.

reg allows the display and modification of all the 6502 registers and the Commodore memory map register. When you open registers with **reg**, you are placed in an interactive mode where all keystrokes are intercepted and handled specially. In **reg**-mode you are able to view each register in turn and modify any one at will.

Output for the **reg** command is in the following general format:

```
register name          aster  size     data

Reg  A                       .byte    $00
Reg  X                       .byte    $FE
Reg  Y                       .byte    $C4
Reg  SP                      .word    $01FD
Reg  PC                      .word    ProgStar
Reg  ST                      .byte    $02
Reg  MM                *     .byte    $30
```

register name is the name of the register:

A	accumulator
X	x-index register
Y	y-index register
SP	stack pointer
PC	program counter

ST	status register
MM	memory map register

aster is a field which contains an asterisk on the currently open register.

size is the size of the data register, either byte or word.

data is the actual data in the register. The data in the PC register will automatically undergo symbol substitution, but the data in the other registers will not unless you request it specifically with the [S] key (see below).

Open reg-mode Keystrokes

When data is opened with the **reg** command, the super-debugger is intercepting keystrokes and responding at that level. When an invalid keystroke or a bad entry is detected, the cursor will briefly flash as a ? symbol. When the cursor is on the asterisk in the *aster* field, the following keystrokes will have an effect:

[SPACE]	enter deposit mode at this location (see deposit description below)
[↓] or [J]	close current register and open next one.
[SHIFT] + [↓] or [K]	close current register and open previous one.
[RETURN]	close current register and return to command prompt.
[.]	display data as decimal.
[$]	display data as hexadecimal.
[?]	display data as octal.
[%]	display data as binary. Note: word values will be displayed in hexadecimal because low/high binary words are seldom useful.
[']	display data as characters.
[S]	display data in symbolic form.

Deposit reg-mode

When you press space at the asterisk prompt, the data field clears and the cursor is placed into it. At this point you can enter new data for this register. As on the command line, DEL deletes the character to the left of the cursor and ← clears the input line.

To enter a line and leave deposit mode, use one of the following keystrokes:

⇆	enter current line and reopen current instruction. (Useful for checking a complex operand expression.)
↓	enter current line and open next register.
SHIFT + ↓	enter current line and open previous register.
RETURN	enter current line and return to command prompt.

If an error is detected in the entry, the line will not be entered and the cursor will briefly flash as a ?.

To leave deposit mode without entering a line, do one of the following:

1. Enter an empty line or a line which contains only spaces.

2. Use DEL to backspace out of the disassembly/deposit field.

reg-mode Deposit Syntax

reg-mode deposits have the following syntax:

exp

If the register size is byte, only the low-byte of the expression will be stored in the register.

The full deposit entry may be up to 100 characters in length. If you try to type beyond the 100 character limit, the cursor will briefly flash as a ?.

Command: **flag**

Synonymn: **fg**

Mini: see **fg** in Chapter 9.

Purpose: open individual flags in the processor status register (ST)

Usage: **flag** [[**f.**]*flagname*]

Note: *flagname* is the optional name of the flag to open (note: a **f.** may be appended to the beginning of the flag name):
n | v | b | d | i | z | c

If no flag is specified, the last flag opened will be reopened.

flag allows the display and modification of all bits in the processor status register (ST, r.st). When you open a flag with **flag**, you are placed in an interactive mode where all keystrokes are intercepted and handled specially. In **flag**-mode you are able to view each flag (bit in the ST register, including the undefined bit 5) in turn and set or clear any one at will.

Output for the **flag** command is in the following general format:

flag symbol	*flag name*	*aster*	*data*
Flag N	Sign (neg.)		%1
Flag V	Overflow		%0
Flag	Undefined		%0
Flag B	BRK flag		%0
Flag D	Decimal mode		%0
Flag I	IRQ disable		%0
Flag Z	Zero flag		%1
Flag C	Carry flag		%1

flag symbol is the common character abbreviation for the flag. The undefined bit (bit 5) has no symbol.

flag name is a descriptive name of the flag.

aster is a field which contains an asterisk on the currently open flag.

data is the actual state of the bit: either set (1) or clear (0).

Open flag-mode Keystrokes

When data is opened with the **flag** command, the super-debugger is intercepting keystrokes and responding at that level. When an invalid keystroke is detected, the cursor will briefly flash as a ? symbol. When the cursor is on the asterisk in the *aster* field, the following keystrokes will have an effect:

SPACE	toggles the state of the flag.
0	clear flag to zero.
1	set flag to one.
↓ or J	close current flag and open next one.
SHIFT + ↓ or K	close current flag and open previous one.
RETURN	close current flag and return to command prompt.

Execution Commands

Command: **go**

Mini: see **go** in Chapter 9.

Purpose: Begin full-speed execution of program.

Usage: **go** [*addrexp*]

Note: *addrexp* is the address to begin execution; if no address is given, execution will begin at the current location of the program counter (PC, r.pc).

The **go** command starts full speed execution of the program. The GEOS screen is displayed and a **jmp** to the proper address is simulated. Control will not return to the super-debugger unless a breakpoint or a **brk** instruction is encountered or the RESTORE key is pressed.

Example:

go ProgStart *begins execution at **ProgStart**.*

go *begins execution at the program counter.*

Command: **runto**

Synonymn: **rt**

Mini: see **rt** in Chapter 9.

Purpose: execute until a given address is reached.

Usage: **runto [*addrexp*]**

Note: *addrexp* is the address where execution will stop. If the *addrexp* is omitted, the current value of the location counter (**u.lc**) will be used.

The **runto** command automates the common debugging procedure of setting a breakpoint, performing a **go** to current location of the program counter, and clearing the breakpoint when control returns to the debugger. If no stop address is specified, the current value of the location counter (**u.lc**) will be used. This allows you to run to the address of the last memory location disassembled.

Example:

runto ProgStart+$1e *sets a breakpoint at **ProgStart+$1e** and executes a **go** to the current location of the program counter.*

Command: **jsr**

Synonymn: **js**

Mini: see **js** in Chapter 9.

Purpose: Execute a subroutine.

Usage: **jsr** ***addrexp***

Note: *addrexp* is the address of the subroutine to execute.

The **jsr** command allows you to execute a subroutine. The super-debugger will simulate a top-step (see **t** command) through an actual **jsr** instruction. The routine at *addrexp* is expected to return with an **rts**.

Example:

jsr SetScreen	*executes a* **jsr** *to the routine at* ***SetScreen.*** *Control returns to the super-debugger when an* **rts** *is encountered.*

Command: **s**

Mini: see **s** in Chapter 9.

Purpose: Single step through instructions and into subroutines.

Usage: **s** [*breakcond*]

Note: *breakcond* is an optional breakpoint condition.

The **s** command will single-step the processor, executing one instruction at a time. The **s** command without any parameters will execute the current instruction pointed at by the program counter (PC, r.pc) and return to the super-debugger, printing the instruction at the new location of the program counter. All processor registers, memory locations, etc. now reflect the results of the instruction just executed. By successively single stepping (pressing ⊡ to repeat the command is good for this), the effects of each instruction may be determined.

The **s** command operates by inspecting the instruction to be executed and determining where the following instruction is located; it then places a temporary breakpoint at that location. For most instructions this is a trivial process because the next instruction to be executed will be the next instruction in memory. However, for instructions which transfer control by reloading the program counter (e.g., **jmp**, **jsr**, **rts**, branches, etc.), the address of the next instruction must be calculated accordingly.

IMPORTANT: You cannot step through ROM . If you try to step through ROM code, you will get an error because breakpoints cannot be set in ROM.

Single-stepping with a Condition

The **s** command will also accept a *breakcond* parameter. Because the **s** command sets a temporary breakpoint for each instruction, the condition will be tested and the counter decremented after every instruction. When you supply a *breakcond*, the following message will be displayed while the super-debugger is stepping:

```
Stepping until condition met...
```

Each time you step through a **jsr** subroutine call, the address of the routine will be added to the step-through-jsr history list. This list gives you an

audit trail of the procedure calls stepped-through as well as allowing the **finish** command. See also: **t**, **history**, and **finish**

If, while stepping with a *breakcond*, a previously set user-defined breakpoint is encountered, the following message will appear:

```
Software breakpoint encountered...
continue (y/n)?
```

You can press [Y] to ignore the breakpoint, or you can press any other key to acknowledge the breakpoint and not step through the instruction.

The s command won't display the GEOS screen unless option 5 is enabled.

Examples:

s .10 *single step through ten instructions.*

s =£f.c *single step until the carry flag is clear.*

s .20,=(r.a == $50 ^^ @buf_flag)
*single-step until the value in the accumulator is equal to $50 or the byte at variable **buf_flag** is non-zero (true) 20 times.*

Command: **t**

Mini: see **t** in Chapter 9.

Purpose: single-step through instructions and top-step through subroutines.

Usage: **t** [*breakcond*]

Note: *breakcond* is the optional breakpoint condition.

The **t** command will single-step the processor, executing one instruction at a time, until it encounters a **jsr** instruction, in which case it will execute the subroutine full speed. The **t** command without any parameters will execute the current instruction or subroutine call pointed at by the program counter (PC, r.pc) and return to the super-debugger, printing the instruction at the new location of the program counter. All processor registers, memory locations, etc. now reflect the results of the instruction or subroutine just executed. Top-stepping is useful for avoiding having to single-step through GEOS routines and already debugged subroutines. It is also useful for executing calls to ROM-based subroutines, which cannot be stepped-through.

The **t** command operates by inspecting the instruction to be executed and determining where the following instruction is located; if the instruction is anything except a **jsr**, it then places a temporary breakpoint at that location as with the **s** command. However, if the instruction is a **jsr**, a breakpoint will be set at the instruction following the subroutine call.

IMPORTANT: You cannot use the **t** command in ROM code; however, you *can* top-step through a **jsr** into ROM. If you try to use the **t** command while in ROM, you will get an error because breakpoints cannot be set in ROM. Also, you should not top-step through GEOS inline subroutine calls (GEOS routines which begin with **i_**); The top-step will set the first byte of the inline data to $00 and the breakpoint will never be encountered. It is best to handle these cases manually. Use **runto** to set a breakpoint at the instruction following the inline data and execute the **jsr**.

Top-stepping with a Condition

The **t** command will also accept a *breakcond* parameter. Because the **t** command sets a temporary breakpoint for each instruction, the condition will be tested and the counter decremented after every instruction. A

subroutine, in this case, is treated as one instruction. When you supply a *breakcond*, the following message will be displayed while the super-debugger is stepping:

```
Stepping until condition met...
```

Encountering User-defined Breakpoints

If, while top-stepping with a *breakcond* or while **t** is executing a subroutine, a previously set user-defined breakpoint is encountered, the following message will appear:

```
Software breakpoint encountered...
continue (y/n)?
```

You can press [Y] to ignore the breakpoint, or you can press any other key to acknowledge the breakpoint and not step through the instruction.

The **t** does not display the GEOS screen unless option 5 is enabled.

Examples:

t .10	*top-step through ten instructions/subroutine calls.*
t =f.c	*top-step until the carry flag is set.*
t u.1,=(r.x >= $10)	*Use the value in user register 1 (**u.1**) and top-step until the X-register is greater than or equal to $10 that many times.*

Command: **p**

Purpose: Proceed until breakpoint encountered.

Usage: **p** [*breakcond*]

Note: *breakcond* is the optional breakpoint condition.

The **p** command will begin execution of code beginning at the current program counter and execute code at full speed until a breakpoint is encountered. When a breakpoint is hit, the super-debugger is given control; if no *breakcond* was specified, the instruction at the current program counter will be printed and you will be returned to the command prompt. If a *breakcond* was specified, the expression is tested and the counter decremented. If the counter reaches zero and the conditional evaluates to true, the breakpoint succeeds. Otherwise the instruction at the breakpoint is executed and the **p** command continues to the next breakpoint.

The **p** command does not display the GEOS screen unless option 5 is enabled.

Using proceed with a conditional allows you to place a breakpoint at the beginning of a subroutine and have the conditional evaluated. This way you can break only if certain special entry conditions exist.

Examples:

p .10 — *proceed until the tenth breakpoint is encountered.*

p =f.d — *proceed until a breakpoint is reached and the decimal mode flag is set.*

p .8,=(@error_flg > disk_err)

*proceed until a breakpoint is encountered. If the variable **error_flag** is greater than **disk_err** eight times, then break.*

Command: **next**

Synonymn: **nx**

Mini: see **nx** in Chapter 9.

Purpose: Set breakpoint at next instruction (physically in memory) and proceed with current instruction.

Usage: **next**

Note: takes no parameters.

The **next** command sets a breakpoint at the next instruction in memory (as opposed to the next instruction to be executed) and proceeds with the current instruction. This command is especially useful for leaving a loop. Most loops consist of a number of instructions followed by a backward branch. Using the **next** command when the program counter is pointing at branch instruction will place a breakpoint at the instruction after the branch and then begins executing with the branch instruction. As long as the branch succeeds and continues looping backwards, execution will continue. When the branch fails, the breakpoint is encountered and control is returned to the command prompt.

IMPORTANT: You cannot use the **next** command in ROM code. If you try to use the **next** command while in ROM, you will get an error because breakpoints cannot be set in ROM.

The **next** command does not display the GEOS screen unless option 5 is enabled.

Example:

Given the following loop:

```
3000 Start                ldx  #$FF
3002 OutrLoop             ldy  #$FF
3004 InnrLoop             dey
3005 InnrLoop+$01         bne  InnrLoop
3007 InnrLoop+$03         dex
3008 InnrLoop+$04 >       bne  OutrLoop
300A InnrLoop+$06         lda  #$00
```

with the program counter at the backward branch at $3008, using the **next** command would place a breakpoint at $300a and execute the branch instruction. When the X-register counts down to $00, the **bne** will fail and the breakpoint will be encountered.

Command: **loop**

Synonymn: **l**

Purpose: Proceed with current instruction and set a breakpoint at the current instruction.

Usage: **loop** [*breakcond*]

Note: *breakcond* is an optional breakpoint conditional.

The **loop** command sets a breakpoint at the current instruction in memory and then proceeds with the current instruction. The breakpoint will be hit before the current instruction is *again* encountered. The idea behind this command is to allow a pass through a loop by waiting for the processor to return to the current instruction. If loop is used without a *breakcond*, the loop will be executed once.

The **loop** command will also accept a *breakcond* parameter. At each pass through the loop (each time the breakpoint is encountered), the conditional is evaluated and the counter decremented. If the conditional evaluates to true and causes the counter to reach zero, the breakpoint succeeds.

IMPORTANT: You cannot use the **loop** command in ROM code. If you try to use the **loop** command while in ROM, you will get an error because breakpoints cannot be set in ROM.

The **loop** command does not display the GEOS screen (unless option 5 is enabled).

Example:

Given the following loop:

```
3000 Start              ldx  #$FF
3002 OutrLoop           ldy  #$FF
3004 InnrLoop    >      dey
3005 InnrLoop+$01       bne  InnrLoop
3007 InnrLoop+$03       dex
3008 InnrLoop+$04       bne  OutrLoop
300A InnrLoop+$06       lda  #$00
```

with the program counter at the **dey** at $3004 in the middle of the loop, using the **loop** command would place a breakpoint at $3004 and proceed with the **dey** instruction. Assuming the X and Y index register are not such that the loop will be exited, the breakpoint will be encountered on the next pass through.

NOTE: The **loop** command is based on **proceed**; you will get strange results if the loop contains any user-defined breakpoints because the *breakcond* will be evaluated at each breakpoint, and not just current location.

Command: **skip**

Purpose: Skip over the current instruction without executing it.

Usage: **skip**

Note: takes no parameters.

The **skip** command increments the program counter to point to the next instruction in memory, causing the current instruction to be skipped over without being executed. The **skip** command is useful through branch instructions which would otherwise succeed or **brk** instructions in your code.

Command: **stopmain**

Synonymn: **sm**

Mini: see **sm** in Chapter 9.

Purpose: Continue program execution until a safe point in the GEOS **MainLoop**, then return to the super-debugger.

Usage: **stopmain**

Note: takes no parameters.

If you use the [RESTORE] key to enter the super-debugger, it is sometimes a good idea to use the **stopmain** command, especially if the processor was in the middle of interrupt code. **stopmain** places a breakpoint in a safe place within the GEOS **MainLoop** and executes a **go**. Assuming the application at hand will return control to mainloop, the breakpoint will be encountered and control will return to the debugger.

For more information on the GEOS **MainLoop**, refer to *The Official GEOS Programmer's Reference Guide.*

IMPORTANT: If you break into the debugger with the [RESTORE] key while interrupt code is being executed and you *do not* do a **stopmain**, a subsequent **getb** or **putb** could destroy a disk.

Stack Related Commands

Command: **stack**

Purpose: displays the top eight bytes on the stack.

Usage: **stack**

Note: takes no parameters.

The stack command looks at the current processor stack (located on page one) and displays the top eight bytes in the following format:

```
Current Stack:
addr    byte word   return address
$01EA   $FF  $04FF  $0500
$01EB   $04  $3004  SetLoop
$01EC   $30  $0630  $0631
$01ED   $06  $6506  $6507
$01EE   $65  $7A65  Do_quit
$01EF   $7A  $EE7A  ClrMenu
$01F0   $EE  $43EE  $43EF
$01F1   $43  $0743  $0744
```

stack starts at the current stack pointer (**sp**, **r.sp**) and progressively reads eight bytes off of the stack. The **addr** field shows the hex location in the stack area on page one. The **byte** field shows the byte value at this location; this is the byte which would be loaded into the accumulator if a **pla** instruction is executed. The **word** field shows the word value at this location (low/high order). The **return address** field shows the address where execution would resume if an **rts** instruction was encountered or a **return** command was executed; it is the word value plus one, and the super-debugger attempts to display it as a symbol.

See also: **return.**

Command: **history**

Synonym: **h**

Purpose: displays the step-through-jsr stack

Usage: **history**

Note: takes no parameters.

Each time the **s** single-step command is used to step through a subroutine call (**jsr**), the address of the routine stepped out of is pushed onto the the super-debugger step-through-jsr stack. The **history** command displays this stack. This gives you an audit trail of how the current point in the program was reached.

The step-through-jsr stack is used in conjunction with the **finish** command. See: **finish** for more information.

Example:

After stepping through a **jsr Graphics** at **$40c** and a **jsr $ca69** at **$c94c**, a history would yield the following display:

```
step through jsr history:
040C ProgStar+$0C          jsr Graphics
C94C $C94C                 jsr $CA69
```

See also: **finish**, **s**, and **inithist**.

Command: **inithist**

Synonymn: **inith**

Purpose: clears the step-through-jsr history.

Usage: **inithist**

Note: takes no parameters.

The **inithist** command completely clears the step-through-jsr history. The history stack is automatically cleared anytime a **go**, **p**, **runto**, **loop**, or **return** command (or any macro based on one of these commands) is issued. The history stack is also cleared when a **brk** instruction is encountered.

Command: **finish**

Synonymn: **fin**

Purpose: finish up (at full-speed) the most recent subroutine that was single-stepped into. Uses the step-through-jsr stack.

Usage: **finish**

Note: takes no parameters.

Each time the **s** single-step command is used to step through a subroutine call (**jsr**), the address of the routine stepped out of is pushed onto the the super-debugger step-through-jsr stack. The **finish** command finishes the most recent subroutine that was single-stepped into, effectively "popping" the newest item on the step-through-jsr stack.

The **finish** command works by checking the step-through-jsr stack and sets a breakpoint at the instruction following the last jsr instruction. **finish** is useful when you accidentally single-step through a **jsr** when you meant to top-step, or if, when checking a subroutine, you are convinced it is not the culprit and want to return to the previous level.

IMPORTANT: You cannot use the **finish** command if it results in trying to set a breakpoint in ROM; you will get an error because breakpoints cannot be set in ROM. Also, the **finish** command should not be used to finish a GEOS inline subroutine calls (GEOS routines which begin with **i_**); The **finish** will set the first byte of the inline data to $00 and the breakpoint will never be encountered.

NOTE: If a software breakpoint is encountered during a top-step and you choose not to continue execution, the top-step's point of entry will be pushed onto the history stack. A subsequent **finish** will continue the top-step, returning to the instruction after the top-stepped **jsr**.

The **finish** command does not display the GEOS screen (unless option 5 is enabled).

Example:

Given the following step-through-jsr history (displayed with the **history** command):

```
step through jsr history:
040C ProgStar+$0C          jsr Graphics
C94C $C94C                 jsr $CA69
```

with the program counter somewhere within the subroutine at $ca69, a **finish** will end up at the instruction following the **jsr $ca69** instruction. A second **finish** will end up at the instruction following the **jsr Graphics** instruction.

See also: **history, inithist, s,** and **return**

Command: **return**

Purpose: run until subroutine returns.

Usage: **return**

Note: takes no parameters.

The **return** command is similar to the **finish** command in that it is designed to run full-speed until the current subroutine is finished. But whereas the **finish** command determines the return address by using the step-through-jsr history, the **return** command determines the return address by using the values on the stack.

return sets a breakpoint at the instruction which will be executed if an **rts** is encountered. It assumes that the top word (two bytes) on the stack are a valid return address. If the subroutine has pushed values onto the stack, **return** will not work correctly.

IMPORTANT: You cannot use the **return** command if it results in trying to set a breakpoint in ROM; you will get an error because breakpoints cannot be set in ROM. Also, the **return** command should not be used to finish a GEOS inline subroutine calls (GEOS routines which begin with **i_**); The **return** will set the first byte of the inline data to $00 and the breakpoint will never be encountered.

See also: **finish** and **stack.**

Breakpoint Commands

When debugging a program, it is often desirable to stop program execution at a specific point so that you can check variables, flags, or registers, making sure they contain correct and expected values. The super-debugger implements this mechanism with *breakpoints*. A user-defined breakpoint, or "breakpoint" for short, can be set at a specific point in the program. When the breakpoint is encountered, control is transferred to the super-debugger, a message is printed and the instruction at the breakpoint is disassembled:

```
*** Software Breakpoint ***
0402 ProgStar+$02b> sta dispBuff
```

The breakpoint is triggered *before* the instruction at the breakpoint is executed. In the above example, the program counter is pointing at the **sta dispBuff** instruction, which is the *next* instruction to be executed.

When a breakpoint is encountered, you can immediately continue execution with the **go** command.

NOTE: You can set up to eight user-defined breakpoints.

How Breakpoints Work: the Nitty Gritty

The 6502 implements a special instruction called **brk** (for *break*), which generates an interrupt. The super-debugger intercepts this interrupt and treats it as a software breakpoint. When you set a user-defined breakpoint, the super-debugger replaces the data byte at the address with a **brk** instruction ($00). By going through the super-debugger, the breakpoints are automatically controlled and managed. Because the super-debugger saves the byte that was replaced, whenever you view, disassemble, or otherwise examine the area from within the super-debugger, the original data will be shown, even though in actuality, the **brk** instruction is in place.

However, there is nothing stopping you from manually placing **brk** instructions in your code by assembling them into your program, either from within the super-debugger or in your source code. These **brk** instructions will actually appear as **brk**'s and will not be managed by the super-debugger. When one of these **brk** instructions is encountered, the super-debugger returns with:

```
BRK instruction encountered...
```

The super-debugger will not execute or step through such a **brk** instruction.

NOTE: Because breakpoints are implemented by modifying data in memory, they cannot be set in ROM. Any attempt to set a breakpoint in ROM will cause a zero to be written to the RAM mapped behind it.

IMPORTANT: be careful setting breakpoints in an overlay module that might get swapped — if you set an automatic breakpoint in an overlay module and then another module is placed over it without first removing the breakpoint, the super-debugger will have no way of knowing the correct module is no longer in memory and could potentially change the wrong code when trying to remove or manage the breakpoint.

Command: **b**

Mini: see **b** in Chapter 9.

Purpose: display currently active breakpoints.

Usage: **b**

Note: takes no parameters

To view the currently active breakpoints, use the **b** command without a parameter. The locations of the currently set breakpoints will be disassembled. For example:

```
0402 ProgStar+$02b      sta dispBuff    first breakpoint
3FCA GRAPH_s3    b      jsr DrawBlk     second breakpoint
5602 Swap    +$42b      sta semaphor    third breakpoint
```

If no breakpoints are set, no lines will be printed.

Command: **setb**

Synonymn: **sb**

Mini: see **sb** in Chapter 9.

Purpose: set a breakpoint

Usage: **setb** [*addrexp*]

Note: *addrexp* is the address where the breakpoint should be set. If and address is not specified, a breakpoint will be set at the address of the current location counter (**u.lc**).

The **setb** command allows you to set a breakpoint in memory. To set a breakpoint at a specific memory location, merely supply an *addrexp* as a parameter. **setb** will evaluate the *addrexp* and set a breakpoint at that location

Example:

setb $4fe	*sets a breakpoint at $4fe*
setb Do_graph	*sets a breakpoint at **Do_graph***
setb @@(r.sp+1)+1	*sets a breakpoint at the address which will be encountered if an **rts** is performed (uses the return address on the stack).*

If you use **setb** without a parameter, a breakpoint will be set at the current address of the location counter (**u.lc**). The location counter is a value maintained by geoDebugger. It holds the address of the most recently opened or displayed memory location. For example, after an **a** command, the location counter points to the address of the last instruction opened. Following an **a** with a **setb** without a parameter would set a breakpoint at this last instruction.

Example:

If the last memory location opened was **$3245,**

setb

would set a breakpoint at this location.

NOTE: It is often easier to set breakpoints with the **a** and **m** open mode commands.

Command: **clrb**

Synonymn: **cb**

Mini: see **cb** in Chapter 9.

Purpose: clear a single breakpoint.

Usage: **clrb** [*addrexp*]

Note: *addrexp* is the address of the breakpoint to clear. If an address is not specified, it will try to clear a breakpoint at the current location counter (**u.lc**).

The **clrb** command allows you to clear a breakpoint in memory. To clear a breakpoint at a specific memory location, merely supply an *addrexp* as a parameter. **clrb** will evaluate the *addrexp* and clear the breakpoint at that location. If there is no breakpoint at that location, the **clrb** command produce an error.

Example:

clrb $4001 *clears a breakpoint at $4001*

clrb Do_graph *clears a breakpoint at **Do_graph***

If you use **clrb** without a parameter, the breakpoint at the current address of the location counter (**u.lc**) will be cleared. The location counter is a value maintained by geoDebugger. It holds the address of the most recently opened or displayed memory location. For example, after a **b** command, the location counter points to the address of the last breakpoint disassembled. Following a **b** with a **clrb** without a parameter clear the last breakpoint listed.

Example:

With the following breakpoint list:

```
0402 ProgStar+$02b      sta dispBuff      first breakpoint
3FCA GRAPH_s3    b      jsr DrawBlk       second breakpoint
5602 Swap    +$42b      sta semaphor      third breakpoint
```

a **clrb** without a parameter would clear the breakpoint at **Swap+$42.**

NOTE: It is often easier to clear breakpoints with the **a** and **m** open mode commands.

Command: **initb**

Synonymn: **ib**

Mini: see **ib** in Chapter 9.

Purpose: initialize (clear) all breakpoints.

Usage: **initb**

Note: takes no parameters

The **initb** command will clear all currently active breakpoints.

Symbol Commands

Command: **sym**

Purpose: Display symbols in currently active modules.

Usage: **sym** [*searchspec*]

Note: Operates on the currently active modules as set with the **set** command. **sym** with no parameter will show all symbols; with a valid *searchspec*, all symbols which match the search-specification will be shown.

To view symbols in the currently active modules, either use the **sym** command without a parameter to view all the symbols or supply a *searchspec* to view all the matching symbols. The symbols will be displayed in the following format, starting with the module which has the highest priority and proceeding in the order established with the **setmod** command:

symbol	*value*	*symbol*	*value*
ProgStar	=$0400	DoQuit	=$0518
PrintBuf	=$7906	isGEOS	=$848B
dblClick	=$8515	year	=$8516
month	=$8517	day	=$8518

Examples:

sym r* *displays all symbols which begin with* ***r****.*

sym ???_x *displays all five-character symbols which end in* ***_x****.*

sym *_* *displays all symbols which contain an underline character.*

sym *displays all symbols.*

Command: **setsym**

Purpose: define or change a symbol in the module with the highest priority.

Usage: **setsym** ***symbol,exp***

Note: Operates on the module with the highest priority as set with the **setmod** command. *symbol* is a valid symbol name and *exp* is the value to equate with the symbol.

To define a new symbol or redefine an existing symbol in the module with the highest priority, use the **setsym** command followed by a valid symbol, a comma, and an expression for the value of the symbol. The symbol will be defined in the module with the highest priority.

Examples:

setsym eric,$4000	*defines a symbol* ***eric*** *with the value* ***$4000****.*
setsym l_data,dBuff+$400	*defines a symbol* ***l_data*** *with the value* ***dBuff+$400****.*

Command: **clrsym**

Purpose: clears (removes) symbols in the currently active modules

Usage: **clrsym** ***searchspec***

Note: Operates on the currently active modules as set with the **setmod** command. **clrsym** with a valid *searchspec*, will delete all symbols in the currently active modules which match the search-specification.

To remove a symbol from the currently active modules, use the **clrsym** command followed by a valid *searchspec*. All matching symbols will be deleted from the currently active modules (as set with the **setmod** command).

NOTE: the **clrsym** command will delete *all* matching symbols from the currently active modules, not just from the module with the highest priority. To delete a symbol from the module with the highest priority, either use the **a** or **m** open commands or deactivate modules with the **setmod** command.

Examples:

clrsym ProgStar	*deletes the symbol **ProgStar** from currently active modules*
clrsym i_*,-	*deletes all symbols which begin with **i_** from the currently active modules*

Command: **initsym**

Purpose: initialize (clear) all symbols in the currently active modules

Usage: **initsym**

Note: Operates on the currently active modules as set with the **setmod** command. Takes no parameters.

The **initsym** command deletes all symbols from all currently active modules.

HINT: To clear all symbols in all modules (not just the active ones) do an **initmod** followed by an **initsym**. The **initmod** enables all the module's symbols and the **initsym** command deletes them.

Command: **mod**

Purpose: display module priority settings.

Usage: **mod**

Note: takes no parameters.

The **mod** command displays the current module priority settings as established with the **setmod** command.

Command: **setmod**

Purpose: Set module symbol table prioritites.

Usage: **setmod [*modlist*]**

Note: *modlist* is a list of module numbers separated by commas; if the last module in the list is an asterisk (*), the remaining modules will be added to the list in numerical order. If no *modlist* is specified, the current module priority will be printed.

When debugging a VLIR application with multiple overlay modules, the super-debugger keeps the symbols for each module in a separate table. Normally, when the super-debugger is searching for a symbol, whether to display it or use it in an expression, it will first search the resident module symbols (module zero) and then the remaining modules in numerical order. It is often desirable, however, to change this search order. You can use the **setmod** command to establish a symbol table priority. This can be used to prevent, say, module five's symbols from showing up while module four is being debugged in memory.

The *modlist* is an ordered list of module numbers. When the symbol table is searched, the super-debugger will search the first table listed, then the second, and so on, until the list is exhausted. If the last module in the list is an asterisk (*), the remaining modules will be added to the list in numerical order. **setmod** * will initialize the search priority to all modules in numerical order beginning with zero (resident). This is equivalent to the **initmod** command.

You can deactivate a module's symbols by leaving it out of the list and not using the * symbol. Symbols in a deactivated module cannot be deleted or displayed without first reactivating them.

NOTE: If you use a nonexistent module number in the *modlist*, the super-debugger will report a command error.

Examples:

setmod 0,3,* *search resident first then module three followed by the remaining tables.*

setmod 0	*only search resident. Deactivate all other modules.*
setmod 5,2,3,0	*search module five, then two, then three, and finally resident (zero). Deactivate all other modules*
setmod *	*equivalent to* ***initmod.***

Command: **initmod**

Purpose: reset module symbol table priorities to the default.

Usage: **initmod**

Note: Takes no parameters.

initmod will initialize the search priority to all modules in numerical order beginning with zero (resident).

Example:

With a five module VLIR file — resident (0), 1, 3, 10, and 11 — an **initmod** will set the module priority to:

0, 1, 3, 10, 11

Macro Commands

The super-debugger is based on a complex macro language. In fact, the commands described in this chapter are special macros called system macros. The macro language allows you to access features of the debugger by simulating the actual keystroke input to the debugger. Virtually everything you can accomplish by typing on the keyboard can be automated in a macro.

Levels of the Macro Language

There are three levels to the macro language: command primitives, system macros, and user-defined macros.

Command Primitives. The lowest and most obscure level of the macro language. A command primitive is an at-sign @ followed by a single character (e.g., @s or @>). All commands decompose into one or more command primitives. For a list of command primitives, refer to Appendix C.

System Macros. All the the super-debugger commands described in this chapter (such as **pc** and **loop**) are actually system macros. System macros are usually composed of one or two command primitives, but many are more complex.

User-defined Macros. Using the **setmac** command or geoWrite, you can create your own macros to either replace or enhance the set of system macros.

How the Super-debugger Parses input

Super-debugger commands are composed of a the actual command followed by up to 10 parameters, separated by commas. When you enter a line at the command prompt, the Super-debugger parser takes the first word (or set of characters) as the command and the remaining text as the parameters. It then searches the list of user-defined macros, looking for a macro which matches the command. If not found, the super-debugger will, in turn, search the list of system macros and finally the list of command primitives. If the the command is not found, the super-debugger prints:

```
*** Command Error ***
```

When the command is found, the super-debugger attempts to execute it.

Arguments

The super-debugger parser assigns each of the possible ten parameters to individual names:

arg0, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8, arg9

In addition, the following names refer to groups of parameters:

all0	first through tenth parameters
all1	second through tenth parameters
all2	third through tenth parameters
all3	fourth through tenth parameters
all4	fifth through tenth parameters
all5	sixth through tenth parameters
all6	seventh through tenth parameters
all7	eighth through tenth parameters
all8	ninth through tenth parameters
all9	tenth parameter

Anywhere either the **arg0-arg9** or **all0-all9** parameters words are used, the super-debugger will make a straight text substitution into the macro. When using the **all0-all9** parameter words, the parameters will be substituted with separating commas.

Creating Macros in geoWrite

You can create macros in geoWrite and have them automatically load into the super-debugger.

The super-debugger looks for two different geoWrite files on the disk:

1: ***appname*.dbm** — where *appname* is the file name of the application being debugged (the **.dbm** is an extender). The super-debugger will look for this file first and load it. (For example, if the application is called **SampleSeq**, the associated debugger macro file will be **SampleSeq.dbm**.)

2: **default.dbm** — this is the defualt debugger macro file. It will be loaded if the super-debugger loaded and **NO FILE** is selected in the file-selection dialog box or if the ***appname*.dbm** file was not found.

When creating a macro in geoWrite, there are a few things to be aware of:

1: A comment can be entered into the macro file in the same way they are enetered into geoAssembler: everything on a line following a semicolon (;) will be ignored.

2: Leading and trailing whitespace is ignored, and a comment at the end of a line is treated as whitespace. The only way to enter a SPACE keystroke into a macro at the beginning or end of a line is to use the special character combination **[sp]**. Spaces within a line will be interpreted correctly, although you can always use the **[sp]** notation.

3: geoWrite carriage returns (at the end of lines) will not be interpreted as presses of the RETURN key. In order to enter a RETURN keystroke, use the special character combination **[cr]**.

4: There are three other keystrokes which must be entered using special character combinations.

keystroke	*notation*
	**[dn]**
SHIFT + 	**[up]**
	**[rt]**
RUN/STOP	**[st]**

For samples of geoWrite macro definitions, refer to the **SampleSeq.dbm** file on your geoProgrammer disk. For information on creating macros within the super-debugger, refer to the **setmac** command.

Autoexec Macro

When the super-debugger first loads, it looks for a macro named **autoexec.** If the macro exists, it is executed. This can be used to set special starting conditions or debugger options automatically at load-time.

Macro Size Limit

The macro table is 1000 bytes in size. This amounts to approximately nine-hundred keystrokes. However, because the input buffer is 100 bytes, the largest macro which can be defined with the **setmac** command is limited by this smaller size. A macro created in geoWrite is not limited by the size of the input buffer and can be as large as 250 keystrokes. If a macro is too large, an error will be shown.

Command: **sysmac**

Synonymn: **sys**

Purpose: display system macros.

Usage: **sysmac** [*searchspec*]

Note: **sysmac** with no parameter will show all system macros; with a valid *searchspec*, all system macros which match the search-specification will be shown.

All geoProgrammer commands are actually system macros — permanent macros programmed in the same macro language as user macros. Most system macros are built-up from command primitives. A command primitive is an at-sign (@) followed by one character. They are the lowest level of control available to macros. By studying how the command primitives are used in the system macros, you can begin using them in your own user-defined macros. Command primitives tend to run faster than their system macro counterparts. (For a list of valid command primitives, refer to Appendix C.) Additionally, there are a few system macros which are used internally by other system macros (e.g., **bkptdo**)

Viewing System Macros

To view system macros, either use the **sysmac** command without a parameter to view all system macros or supply a *searchspec* to view all system macros whose names match the search specification. The macros will be displayed in the same format as the **mac** command.

Examples:

sysmac skip *displays the system macro **skip**.*

sysmac ?? *displays all system macros whose names have only two characters.*

As an example, we will decipher the **skip** system macro (command) to understand how it works. If you were to do a **sysmac skip**, you would see the following:

```
Macro    Definition
skip.....@0[cr]
         @/r.pc[cr]
         j>[cr]
         @h[cr]
         @>[cr]
```

`@0[cr]`
The **@0** command primitive controls screen output. It is used by the **poff** command to disable screen printing. The **@0** here is equivalent to a **poff**. The carriage return enters the command.

`@/r.pc[cr]`
The **@/** is used to enter the **a** command's open mode (open a memory location as assembly language). With the parameter **r.pc**, we are opening the current location of the program counter, or the instruction we wish to skip. The carriage return enters the command.

`j>[cr]`
Since the previous command placed us into an open mode, these macro keystrokes will be interpreted as open-mode keystrokes. The **j** is equivalent to pressing , which opens the next instruction, and the **>** places the program counter at this new location. This has the effect of skipping over one instruction without executing it. The carriage return leaves the open mode.

`@h[cr]`
The **@h** command primitive does the opposite of the **@0** primitive. It reenables printing. This is equivalent to the **pon** command. The carriage return enters the command.

`@>[cr]`
The **@>** is used by the **pc** command. Here we are giving it no parameter, so the current location of the program counter will be disassembled to the screen. This is equivalent to using **pc** without a parameter. The carriage return enters the command.

Command: **mac**

Purpose: display or remove user-defined macro

Usage: **mac** [*searchspec*]

Note: **mac** with no parameter will show all user-defined macros; with a valid *searchspec*, all user-defined macros which match the search-specification will be shown.

To view user-defined macros, either use the **mac** command without a parameter to view all user-defined macros or supply a *searchspec* to view all user-defined macros whose names match the search specification.

Examples:

mac mymac *displays the user-defined macro* ***mymac****.*

mac ?? *displays all user-defined macros whose names have only two characters.*

The macros will be displayed in the following format:

```
Macro    Definition
sr........s all0[cr]
           pr[cr]
           r[cr]
           pr "------------"[cr]
```

The name of the macro is printed flush against the left edge of the display under the Macro heading. The name is followed by a string of ellipses, and the definition is displayed tabbed out under the Definition heading. Some keystrokes undergo translation:

Key	*Displayed as*	*Description*
RETURN	[cr]	*carriage return*
↕	[dn]	*down*
SHIFT + ↕	[up]	*up*

→	[rt]	*right*
RUN/STOP	[st]	*stop*
SPACE	[sp]	*space*†

†Only leading space characters are translated to [sp].

Command: **setmac**

Purpose: create a user-defined macro

Usage: **setmac** ***name***

Note: *name* is a valid macro name. A macro name can be any combination of letters, numbers, and the underscore symbol.

The **setmac** command creates user-defined macros and requires the macro name as a parameter. The macro name can be any length and may contain any combination of letters, numbers, and the underscore symbol. The following are valid macro names:

123_print
_test
show
R_E_S_E_T

If you create a user-defined macro with the same name as a system macro, the user-defined macro will take precedence. This allows you to redefine any of the system commands to suit your preferences.

IMPORTANT: Be careful — if you redefine the **mac** command, you will be unable to view or delete any macros. If you accidentally do this, type

@;mac

which will use the **clrmac** command primitive to delete the erroneous **mac** definition.

If you create a user-defined macro with the same name as another user-defined macro, the old defintion will be replaced by the new one.

Creating the Macro

When you use the **setmac** command, the following appears:

```
Enter commands. Press <STOP> to end.
```

Most keystrokes you enter at this point will become part of the macro definition. Some keystrokes are used by the **setmac** command and so cannot be entered into a the macro:

INST/DEL	Deletes the character to the left of the cursor and backspaces.
←	Deletes all text up to the last carriage return.
RUN/STOP	Ends the macro definition and returns to the command prompt.

In addition, some keystrokes will be translated into special character combinations to improve the readability of the macro:

↓	`[dn]`
SHIFT + ↓	`[up]`
→	`[rt]`

NOTE: When defining macros within the super-debugger, you cannot use the special keystroke translations (e.g., **[cr]**), as you can when creating macros in geoWrite, and expect them to be converted into the proper keystroke. The super-debugger would interpret **[cr]** as a series of four keystrokes.

Command: **clrmac**

Purpose: remove user-defined macros.

Usage: **clrmac** *searchspec*

Note: *searchspec* is a valid search-specification for the macro's name. All user-defined macros which match the search-specification will be deleted.

The **clrmac** command is used to delete specific macros. All user-defined macros whose names match the search-specification will be deleted.

clrmac sample *deletes the user-defined macro* ***sample****.*

clrmac sam* *deletes all user-defined macros whose names begin with* ***sam****.*

Command: **initmac**

Purpose: initialize (delete) all user-defined macros.

Usage: **initmac**

Note: takes no parameters.

The **initmac** command deletes all user-defined macros.

Command: **poff**

Purpose: Turn off screen printing

Usage: **poff**

Note: Takes no parameters

The **poff** command disables screen printing. A macro can use **poff** to hide much of its operation, thereby avoiding screen clutter and providing an overall cleaner display. **poff** is used in combination with the **pon** command. Where **poff** disables screen printing, **pon** will re-enable it. Printing is automatically enabled whenever a macro returns to the command prompt.

Command: **pon**

Purpose: Turn on screen printing

Usage: **pon**

Note: Takes no parameters

The **pon** command is used in combination with the **poff** command. Where **poff** disables screen printing, **pon** will re-enable it. Printing is automatically enabled whenever a macro returns to the command prompt. **pon** is only useful in the context of a macro.

Command: **if**

Purpose: macro conditional.

Usage: **if *exp,macname***

Note: *exp* is a valid expression (usually a logical expression) and *macname* is the macro or command to execute if the expression evaluates to true (non-zero).

Although the **if** command can be used outside of a macro, it is especially useful within macros because it allows the macro to dynamically base its action upon the evaluation of some expression.

When the **if** command is encountered, the expression is evaluated. If the expression is true (non-zero), the current macro is suspended and the macro specified in the **if** is executed; when the conditional macro is done and ends normally (without the **stop** command), execution of suspended macro continues. If the expression is false, the current macro continues without executing the conditional macro.

Example:

if (u.lc>=Strt && u.lc<=End),pr "in data structure"
If the location counter is somewhere in data area, print a message.

Command: **for**

Purpose: looping command.

Usage: **for *range,macname***

Note: *range* is a valid range and *macname* is the macro or command to execute for each value in the range (the range is inclusive).

The **for** command is analagous to the BASIC **for-next** loop: it allows a command or macro to be executed any number of times, using a counter (**u.fn**) to hold the current value of the range. The **for** command can be used from the command line, but it is especially useful within macros.

As an example:

for .1:.10,dis

would display 10 screens of disassembled code using the **dis** command.

The **for** command uses the **u.fn** user register to maintain the current value of the counter. This allows you to use the value of the counter in an expression. Note, however, that because there is only one **u.fn** register, you cannot nest for loops. That is: a **for** loop which calls a macro which contains another **for** loop will not operate correctly.

Command: **stop**

Purpose: stop macro execution and return to the command prompt.

Usage: **stop**

Note: takes no parameters.

The **stop** command provides a general-purpose way to abort a macro. Anytime a **stop** is encountered, control is immediately returned to the command prompt.

Memory Commands

Command: **find**

Purpose: Find a pattern in memory.

Usage: **find** ***findrange,exp*[*,exp*]**

Note: *findrange* is either a standard *range* which establishes the range of memory which will be searched, or an asterisk (*) which establishes all of memory as the search range ($0000:$ffff). The expression list following the range is a pattern of bytes to search for; any value larger than one byte (>$ff) will be truncated to a byte.

The **find** command searches through memory looking for a specific pattern of byte values. Each time the pattern is found, the location of the first byte of each instance found will be displayed in the same format as the **m** command.

Examples:

find DataBuf:#2000,%11001001,'a'&$80

*Searches 2000 bytes beginning at **DataBuf**, looking for a binary %11001001 followed by an ASCII "a" with the high bit set.*

find in_strng+1:#@(in_string),NULL

*Searches a string input buffer looking for the end of the string as marked by a **NULL** ($00); the first byte of the input buffer holds the maximum length of the string.*

To search for word values, use the low-byte and high-byte operators. For example, the following will find a instances of the address **ProgStar** in the range $3000-$4000 (remember, words are stored in low/high order):

find $3000:$4000,[ProgStar,]ProgStar

To search for strings, use a list of characters. For example, the following will find instances of the word "disk" in all of memory:

find *,'d','i','s','k'

Command: **fill**

Purpose: Fill memory with a pattern

Usage: **fill *range,exp*[*,exp*]**

Note: *range* establishes the range of memory which will be filled. The expression list following the range is a pattern of bytes to fill with; any value larger than one byte (>$ff) will be truncated to a byte.

The **fill** command deposits a specific pattern of byte values into a range of memory.

Examples:

fill buffer:buff_end,0 *clears a buffer to all zeros.*

To fill with word values, use the low-byte and high-byte operators. For example, the following will deposit the word value **FillWord** in the range $3000-$4000 (remember, words are stored in low/high order):

fill $3000:$4000,[FillWord,]FillWord

To fill with a string, use a list of characters. For example, the following will fill the range $3000-$4000 with the word "GEOS"

fill $3000:$4000,'G','E','O','S'

NOTE: When filling with a pattern of more than one byte, the fill command will never exceed the given range, even if the pattern is incomplete when it stops.

Command: **copy**

Purpose: copies a block of memory from one location to another

Usage: **copy *range,addrexp***

Note: *range* is the range of memory to copy from and *addrexp* is the starting address of the copy destination.

The **copy** command copies a block of memory from one area to another. The source is unaffected unless the areas overlap, in which case the source will partially overwrite itself. The **copy** command is intelligent enough to avoid overwriting bytes in the source before they are copied.

Examples:

copy testdata:#datalen,datstruct

*copies some test data (of length **datalen**) into a data structure.*

copy $00:$ff,$3000

copies all of zero page to $3000

copy $600:.100,$608

copies 100 bytes at $600 to $608, overwriting the source (essentially, moving the first 100 bytes up 8 bytes).

Command: **diff**

Purpose: compares two blocks of memory, byte-by-byte.

Usage: **diff *range,addrexp***

Note: *range* is the range of memory to compare from (source) and *addrexp* is the starting address of the range of memory the first range will be compared against (destination).

The **diff** commmand does a byte-by-byte compare between two blocks of memory. Any differences are displayed in the following format:

```
Source              Destination
 0200  .byte $60  <->  040A  .byte $FF
 0210  .byte $00  <->  041A  .byte $A1
```

The mismatching byte at the source is displayed first, followed by the mismatching byte at the destination.

Examples:

diff $1000:$1fff,$2000 — *compares the range of memory from $1000 to $1fff with the range at $2000 to $2fff.*

diff r0l,r1l — *compares the byte at **r0l** with the byte at **r1l**.*

HINT: If you keep all your variables together, you can find which are modified by a given routine by first using the **copy** command to copy them to a free area in memory, calling the routine, and then using the **diff** command to find out which ones have changed. For example:

First make a copy of the variable space:
copy VarStart:VarEnd,FreeRAM

Then execute the routine. When it returns compare the two blocks:
diff VarStart:VarEnd,FreeRAM

Special Commands

Command: **setu**

Purpose: set user variable.

Usage: **setu [u.]*ureg,exp***

Note: *ureg* is the user register to set (note that the **u.** is optional):
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | lc | ws | wc
exp is the word value to store in the register.

The **setu** command is the only way to set a user register. Be careful when changing the **u.lc**, **u.ws**, and **u.wc** counters as the super-debugger uses them extensively.

Examples:

setu u.0,r.sp — *sets user register 0 to the word value of the stack pointer.*

setu 7,@(u.1) — *sets user register 7 to the byte value pointed to by the address in user register 1.*

IMPORTANT: Changing the value of **u.ws** (window size) can lead to unpredictable results. Currently this value is a constant 24, but in future implementations this may change.

HINT: In a macro, setting the **u.wc** register to the value of the **u.ws** register (**setu u.wc,u.ws**) will cause the "MORE" prompt to be displayed after the next line which is printed. This won't work outside of a macro because the **u.wc** register is reset when control returns to the command prompt.

Command: **pc**

Mini: see **pc** in Chapter 9.

Purpose: view or set program counter (**PC**, **r.pc**).

Usage: **pc** [*addrexp*]

Note: *addrexp* is the address to set the program counter at; the new address of the program counter will be disassembled to the screen. If an address is not specified, the current value of the program counter will be disassembled to the screen.

The **pc** command is a quick and easy way to set the program counter. As a side benefit, the **pc** command (with or without a parameter) will also set the location counter (**u.lc**) to the address of the program counter, thereby causing a subsequent command which uses that value, such as **dis** or **a**, to begin at the program counter.

Command: **rboot**

Purpose: reboot GEOS.

Usage: **rboot**

Note: takes no parameters.

If GEOS becomes corrupted during debugging, a standard **quit** would likely crash the system, leaving no alternative but a power-down. The **rboot** command attempts to reboot GEOS and return to the deskTop, thereby salvaging the current contents of the RAM-expansion unit and saving the time necessary to reboot GEOS from BASIC. Before actually leaving, you will asked you confirm your intention to rebot:

```
Reboot GEOS (y/n)?
```

Typing [Y] will exit; typing [N] or any other key will return to the command prompt.

rboot is a last-ditch effort to save the system and should not be used as an everyday alternative to the **quit** command.

Disk Commands

Command: **drivea, driveb, disk**

Synonynm: **da** for **drivea**, **db** for **driveb**, and **di** for disk.

Mini: see **da**, **db**, and **di** in Chapter 9.

Purpose: set or determine current drive.

Usage: **drivea**
driveb
disk

Note: takes no parameters.

The **drivea** and **driveb** commands open the disk in drive A or drive B, respectively, and make that drive the current drive. Subsequent disk commands will access the current drive. **disk** merely shows the current drive and the name of the disk in the current drive. These commands call the GEOS **SetDevice** and **OpenDisk** routines.

Command: **dir**

Purpose: Display directory of the disk in the current drive.

Usage: **dir**

Note: takes no parameters.

The **dir** command shows the directory of the disk in the current drive. The current drive is set with the **drivea** or **driveb** command. The directory display is in the following format:

```
Filename        Track Sector
SamSeq.......... $01   $10
SamSeqHdr....... $04   $06
SamSeq.lnk...... $10   $0B
MyData.......... $0A   $07
```

dir uses a number of system macros, one of which, **fileinfo**, deserves special mention. **fileinfo** is executed each time a directory entry needs to be printed. When called, user-register nine (**u.9**) is an address pointing to a valid directory entry. The directory entry is a 32-byte entity which is described in detail in *The Official GEOS Programmer's Reference Guide*. By defining a your own version of **fileinfo** as a user macro, you can customize the way the directory entries are displayed. Whenever **dir** calls **fileinfo**, your version will take precedence.

Command: **getb**

Synonymn: **gb**

Mini: see **gb** in Chapter 9.

Purpose: get block from the disk in the current drive.

Usage: **getb** [*track,sector*]

Note: *track* is a valid track number *exp* and *sector* is a valid sector number *exp* for the current drive. If the track and sector are not provided, the values in the GEOS **r1L** and **r1H** registers will be used.

The **getb** command reads one sector from the current drive into **diskBlkBuf** at $8000 and then executes a **dumpd** command to display the sector. The values of the track and sector number read will be left in **r1L** and **r1H**; a subsequent **putb** could then be used to write out the sector just read. **getb** calls the GEOS **GetBlock** routine.

getb $12,$0 *get the first block of the directory.*

Command: **putb**

Synonymn: **pb**

Mini: see **pb** in Chapter 9.

Purpose: put a block to disk in the current drive.

Usage: **putb** [*track,sector*]

Note: *track* is a valid track number *exp* and *sector* is a valid sector number *exp* for the current drive. If the track and sector are not provided, the values in the GEOS **r1L** and **r1H** registers will be used.

The **putb** command writes one sector from **diskBlkBuf** at $8000 to the disk in the current drive. The values of the track and sector number written will be left in **r1L** and **r1H**. **putb** calls the GEOS **PutBlock** routine.

IMPORTANT: Be careful using **putb**, especially with no parameters; it is very easy to destroy a disk by writing to the wrong track and sector, especially if **r1L** or **r1H** contain bad values.

Example:

putb .15,.5 put a block at track 15, sector 5.

HINT: **getb** and **putb** can be used together. You can read in a specific sector with **getb**, modify it in **diskBlkBuf** (without affecting **r1L** and **r1H**) and then write it back out again by using **putb** with no parameters.

Command: **getn**

Purpose: get next logical block from disk in current drive.

Usage: **getn**

Note: takes no parameters.

The **getn** command uses the link (track, sector) information of the block currently in **diskBlkBuf** ($8000, $8001) to do a **getb** for the next logical sector in the chain. This command assumes there is a valid sector in **diskBlkBuf.**

Command: **getchain**

Purpose: get and display a chain of sectors from the current drive

Usage: **getchain** [*track,sector*]

Note: *track* is a valid track number *exp* and *sector* is a valid sector number *exp* for the current drive. If the track and sector are not provided, the values in the GEOS **r1L** and **r1H** registers will be used.

The **getchain** command combines the **getb** and **getn** commands. It successively reads in displays blocks which are logically linked. If the track and sector are not supplied, the values in **r1L** and **r1H** will be used. **r1L** and **r1H** are set to valid track and sector numbers by **getb** and **putb**. The values of the track and sector numbers of the last block read will be left in **r1L** and **r1H**.

Example:

getchain .15,.1 *gets a chain of blocks beginning at track 15, sector 1.*

Command: **dumpd**

Synonymn: **dd**

Mini: see **dd** in Chapter 9.

Purpose: dump disk block buffer (**diskBlkBuf**)

Usage: **dumpd**

Note: takes no parameters.

The **dumpd** command dumps all 256 bytes of the disk block buffer (**diskBlkBuf**) at address $8000 in the standard **dump** format.

Chapter 9: Mini-debugger Reference

If you do not have a RAM-expansion unit connected to your Commodore, or you hold down the RUN/STOP key while geoDebugger is loading, geoDebugger will automatically configure itself as a mini-debugger. The mini-debugger is a subset of the super-debugger and supports many of the more useful commands. The biggest difference between the mini-debugger and the super-debugger is the absence of expressions, macros, and symbols. With these limitations in mind, you will quickly understand how the mini-debugger commands correspond to their usually more powerful super-debugger counterparts. (For more information on the super-debugger, refer to Chapter 8.)

This is a reference chapter for the mini-debugger configuration of geoDebugger. Because many of the elements of the mini-debugger are similar or identical to elements in the super-debugger, many descriptions refer the reader to the appropriate sections in Chapter 8.

Although this is primarily a reference chapter, it would be a good idea to read it through completely at least once. For a general overview and information on using the mini-debugger from the GEOS deskTop, refer to Chapter 7.

Memory Usage

The mini-debugger resides entirely in RAM just below the background screen buffer. Because of this you cannot use any memory in the range $3e00 to $5fff. Be sure that your psect and ramsect data areas do not extend into this region.

Case Sensitivity

The mini-debugger is case-insensitive. You may type commands and hexadecimal letters in upper- or lower-case, or any mixture thereof, and the debugger will interpret them correctly. There is one exception: when depositing ASCII strings with the **m** open-mode command, the letter case is maintained; a lower-case "a" will be deposited as a lower-case ASCII value.

Expressions and Numeric Constants

The mini-debugger does not have a complex expression evaluator like the super-debugger, and the only numerical expressions it understands are hexadecimal numbers. You cannot add, subtract, multiply, or perform logical operations.

The hexadecimal numbers consist of an optional dollar sign ($) followed by a string of hexadecimal digits (0-9, a-f). The number cannot exceed 16-bits which limits it to a maximum of four hexadecimal digits.

Examples: **$4f9c**
fff
3ca4
$c

NOTE: When assembling code, a lone **A** or **a** as in **lsr a** will be interpreted as accumulator addressing mode as opposed to a **$a**; use the **$** radix symbol to avoid this confusion.

Basic Operation

The Command Prompt

The basic geoDebugger command prompt is a greater-than (>) symbol in the leftmost column at the bottom of the screen. Whenever this prompt is displayed, geoDebugger is idle, awaiting a command. You can type commands in at this point. The following keystrokes have an effect in this mode:

RETURN	Enters the current line; the super-debugger will attempt to interpret and process the command.
DEL	Deletes the character to the left of the cursor.
←	Erases the current input line.

[,]	Reprints the last command on the current input line, which allows the command to be edited and then re-entered with [RETURN]. The comma must be typed as the first character on the input line.
[.]	Repeats the last command. This is similar to pressing [,] followed by [RETURN]. The period must be typed as the first character on the input line.

These keystrokes are identical to those found in the super-debugger.

Hot Key Entry and Cancel

When your program is running, the [RESTORE] key acts as a "hot key"; it will suspend execution and enter the debugger. When you are in the mini-debugger, [RESTORE] will cancel a command and return to the input prompt at any time. Because of a hardware limitiation in the Commodore keyboard, you may have to press [RESTORE] a couple of times to get it to respond.

The More Prompt

When a command fills the screen with text, the print routine will pause and display a "more" prompt. At the prompt you can press the space bar to get another full screen of text or you can press [RETURN] to get just one more line.

[SPACE]	full screen of text.
[RETURN]	one more line.

Viewing the GEOS Application Screen

You can switch between the mini-debugger text screen and the GEOS application's hi-res screen any time the > command prompt is displayed and the cursor is in the first column by the pressing the [F7] key. This is different from the super-debugger which lets you view the application's screen at any time, not just at the command prompt. You can return to the debugger screen by pressing any other key.

NOTE: In order to save memory, whenever you view the application screen the mini-debugger will clear all but the most recently printed line from the debugger screen.

EnterDeskTop Vector Trap

The mini-debugger sets an permanent breakpoint at the GEOS **EnterDeskTop** vector. If an application attempts to exit by calling **EnterDeskTop**, the following will be printed:

```
*** EnterDeskTop vector encountered ***

C22C 00          >brk
```

When the mini-debugger is running, an application cannot be allowed to exit to the deskTop directly. geoDebugger must first remove itself in order for the deskTop to function properly. To return to the deskTop, use the mini-debugger **q** (quit) command.

Mini-debugger Command Summary

General Commands

q	Exits geoDebugger and returns to the deskTop.
g0	Disable GEOS screen during stepping.
g1	Enable GEOS screen during stepping.

General Display Commands

r	Display processor registers.
d	Display a block of memory in hex and ASCII format.
w	Disassemble a window of code from program counter down.

Open Modes (register and memory examination and modification)

a	Open memory as assembly language code.
m	Open memory as data.
rg	Open processor registers.
fg	Open processor status register as individual flags.

Execution Commands

go	Start full speed execution of program.
rt	Set breakpoint and go.
js	Execute subroutine at address (perform a **jsr**).
s	Single-step through current level and subroutines.
t	Single-step through current level and top-step through subroutines.
nx	Proceed until next instruction is reached (for exiting loops).
sm	Stopmain; stop execution in GEOS **MainLoop**.

Breakpoint Commands

b	Display breakpoints.
sb	Set a breakpoint.
cb	Clear a breakpoint.
ib	Initialize breakpoint table, clearing all breakpoints.

Special Commands

pc	View and set program counter.

Disk Commands

da	Make drive A the current drive.
db	Make drive B the current drive.
di	Display name of disk in current drive.
gb	Get disk block from current drive.
pb	Put disk block to current drive.
dd	Display disk buffer in hex and ASCII format.

Syntax Notation

The following conventions are used in the syntax descriptions of the super-debugger commands. Much of this notation will be familiar from geoAssembler and geoLinker.

hexconst	a hexadecimal constant: a one or two byte number composed of hexadecimal digits optionally preceded by a $.

string	a string of ASCII characters enclosed in double-quotes.
[ ]	square brackets indicate an optional item which may appear zero or one times.
{ }	curly braces indicate an optional item which may appear one or more times.
|	a vertical line indicates a choice and can be read as "or"

In addition, all sample output from the mini-debugger will be printed in a `bold courier` font so that the spacing will closely match the standard Commodore text mode.

General Commands

Command: **q**

Synonymn: **q, e**

Super: see **quit** in Chapter 8.

Purpose: leave the mini-debugger and return to the GEOS deskTop.

Usage: **q**

Note: takes no parameters.

q leaves the mini-debugger and returns to the GEOS deskTop by disabling itself and performing a standard application exit (calls **EnterDeskTop**). The program space will be cleared. If GEOS was corrupted during the debugging session (trampling the memory from $c000 to $a000 is a great way to do this), **q** will very likely crash the system, leaving you no alternative but to reboot by turning off the power. Unlike the super-debugger, the mini-debugger does not support the **rboot** command.

Before actually leaving, you will be asked you confirm your intention to quit:

```
Exit to deskTop (y/n)?
```

Typing [Y] will exit; typing [N] or any other key will return to the command prompt.

Command: **g0, g1**

Super: see **opt** in Chapter 8.

Purpose: enable/disable GEOS screen during stepping.

Usage: **g0**
g1

Note: Takes no parameters.

Normally, when stepping with the mini-debugger **s**, **t**, or **nx** commands, the application's GEOS screen is not displayed. The **g1** command will enable the GEOS screen, causing it to be displayed while stepping. To again disable the GEOS screen, use the **g0** command.

g0 is equivalent to the super-debugger **opt 5,0.**
g1 is equivalent to the super-debugger **opt 5,1.**

Display Commands

Command: **r**

Super: see **r** in Chapter 8.

Purpose: display processor registers.

Usage: **r**

Note: takes no parameters.

The **r** command displays all the processor registers, including the MM (memory map) pseudo-register. The output is identical to the super-debugger **r** command (refer to **r** in Chapter 8).

See also: **rg** and **pc**.

Command: **d**

Mini: see **dump** in Chapter 8.

Purpose: dumps 128 ($80) bytes to the screen in hexadecimal and ASCII.

Usage: **d** [*hexconst*]

Note: *hexconst* is the starting address for the dump. If no address is specified, the value of the current location counter will be used.

d is used to view 128 bytes of memory at once. The ouptut is identical to the super-debugger **dump** command (refer to **dump** in Chapter 8).

Command: **w**

Super: see **w** in Chapter 8.

Purpose: disassembles a window of code at the program counter. Displays five lines of code, starting with the current program counter location.

Usage: **w**

Note: takes no parameters.

The **w** command disassembles five lines of code beginning with the current program counter. It useful for seeing the instructions about to be executed. The output is in the standard disassembly format as described under the mini-debugger **a** command.

Example:

With the program counter at $0404, a **w** might produce the following output:

```
hex
address   hex codes        disassembly

0404      A9 04     >      lda #$04
0406      85 03            sta $03
0408      A9 28            lda #$28
040A      85 02            sta $02
040C      20 36 C1         jsr $C136
```

See also: **pc**.

Open Modes

Command: **a**

Super: see **a** in Chapter 8.

Purpose: open memory for assembly language code.

Usage: **a** [*hexconst*]

Note: *hexconst* is the address to open. If no parameter is specified, the current address pointed to by the location counter will be opened.

a is the general disassemble, assemble, and modify open command. The mini-debugger version of **a** operates much like the super-debugger version.

When you open a memory location with **a**, you are placed in an interactive mode where all keystrokes are intercepted and handled specially. In **a**-mode you are able to disassemble code forward and backward and modify instructions at any point.

Output for the mini-debugger **a** command is in the following general format:

```
hex
address hex codes          aster disassembly

0400 A9 C0          >      lda #$C0
0402 85 2F                 sta $2F
0404 A9 04                 lda #$04
0406 85 03                 sta $03
0408 A9 28                 lda #$28
```

hex address is the absolute address of the instruction. Instructions are either one, two, or three bytes in length.

hex codes displays the hexadecimal bytes which comprise the instruction, as in the following example:

```
0408 A9 28          lda #$28
```

where **A9** is the hexadecimal value for **lda** immediate, and **28** is the hexadecimal value for **#$28.**

flag is a field with three positions, each of which has a unique possible symbol:

b	breakpoint set at this instruction.
>	program counter points at this instruction.
*	Current opened instruction.

disassembly is a disassembly of the bytes at the address. If the location does not contain a valid 6502 opcode, `???` will be displayed.

Mini-debugger Open a-mode Keystrokes

When memory is opened with the **a** command, the mini-debugger is intercepting keystrokes and responding at that level. When an invalid keystroke or a bad entry is detected, the cursor will briefly flash as a `?` symbol. When the cursor is on the asterisk in the *flag* field, the following keystrokes will have an effect:

SPACE	enter deposit mode at this location (see deposit description below)
↓ or J	close current instruction and open next instruction.
SHIFT + ↓ or K	close current instruction and open previous instruction.
←	reopen current instruction.
RETURN	close current instruction and return to command prompt.
M	switch from **a**-mode to **m**-mode. See: **m** command.
B	set breakpoint at this address.
>	set program counter to this address.

Deposit a-mode

When you press space at the asterisk prompt, the disassembly field clears and the cursor is placed into it. At this point you can enter a new 6502 instruction. As on the command line, DEL deletes the character to the left of the cursor and ← clears the input line.

To enter a line and leave deposit mode, use one of the following keystrokes:

⇆	enter current line and reopen current instruction.
↕	enter current line and open next instruction.
SHIFT + ↕	enter current line and open previous instruction.
RETURN	enter current line and return to command prompt.

If an error is detected in the entry, the line will not be entered and the cursor will briefly flash as a ?.

To leave deposit mode without entering a line, do one of the following:

1. Enter an empty line or a line which contains only spaces.

2. Use DEL to backspace out of the disassembly/deposit field.

a-mode Deposit Syntax

The a-mode deposit entry must be a valid 6502 opcode/operand construction as in geoAssembler. Because the mini-debugger does not support expressions or any radix other than hexadecimal, any numbers in the operand must conform to this limitation. Also: you cannot type 40 characters. If you exceed this limit, the cursor will briefly flash as a ?.

Example deposit entries:

lda #$fe	*opcode and hexconst immediate value.*
sei	*opcode alone.*
jsr 33ef	*opcode and hexconst address.*

Command: **m**

Super: see **m** in Chapter 8.

Purpose: open memory for data.

Usage: **m** [***hexconst***]

Note: *hexconst* is the address to open. If no parameter is specified, the current address pointed to by the location counter will be opened.

m is the general view and modify data command. The mini-debugger version of **m** operates much like the super-debugger version.

When you open a memory location with **m**, you are placed in an interactive mode where all keystrokes are intercepted and handled specially. In **m**-mode you are able to view data forward and backward and modify it at any point.

Output for the **m** command is in the following general format:

```
hex
address hex code          aster mode     data

046B 53                       .byte   $53
046C 61                       .byte   $61
046D 6D                       .byte   $6D
046E 70                       .byte   $70
046F 6C                       .byte   $6C
0470 65                 *     .byte   $65
```

hex address is the absolute address of the data.

hex code is the hexadecimal bytes which comprise the data, as in the following examples:

```
0470 65                       .byte   $65
046D 6D 70                    .word   $706D
```

flag is a field with three positions, each of which has a unique possible symbol:

b	breakpoint set at this instruction.
>	program counter points at this instruction.
*	Current opened instruction.

mode is the data display mode, either `.byte` or `.word`. Data shown in word format is displayed in low/high order as in the following example:

data is the actual data at the current address. The data will not undergo symbol substitution unless you request it specifically with the [S] key (see below).

Mini-debugger Open m-mode Keystrokes

When data is opened with the **m** command, the super-debugger is intercepting keystrokes and responding at that level. When an invalid keystroke or a bad entry is detected, the cursor will briefly flash as a ? symbol. When the cursor is on the asterisk in the *flag* field, the following keystrokes will have an effect:

[SPACE]	enter deposit mode at this location (see deposit description below)
↕ or [J]	close current location and open next one.
[SHIFT] + ↕ or [K]	close current location and open previous one.
⇆	reopen current location (useful for rereading a hardware location with changing values).
[RETURN]	close current location and return to command prompt.
[D] [B]	display data as byte.
[D] [W]	display data as word.
[A]	switch from **m**-mode to **a**-mode. See: **a** command.

B	set breakpoint at this address. (not extremely useful when looking at data.)
>	set program counter to this address. (not extremely useful when looking at data.)

Deposit m-mode

When you press space at the asterisk prompt, the data field clears and the cursor is placed into it. At this point you can enter new data for this address. As on the command line, DEL deletes the character to the left of the cursor and ← clears the input line.

To enter a line and leave deposit mode, use one of the following keystrokes:

↔	enter current line and reopen current instruction.
↕	enter current line and open next instruction.
SHIFT + ↕	enter current line and open previous instruction.
RETURN	enter current line and return to command prompt.

If an error is detected in the entry, the line will not be entered and the cursor will briefly flash as a `?`.

To leave deposit mode without entering a line, do one of the following:

1. Enter an empty line or a line which contains only spaces.

2. Use DEL to backspace out of the disassembly/deposit field.

m-mode Deposit Syntax

m-mode deposits for `.byte` and `.word` deposits is slightly different:

`.byte` *string* | *hexconst*{,*hexconst*}

You cannot deposit more than 10 bytes (10 characters or 10 values) in a single deposit. Each *hexconst* must be a one-byte value ($00-$ff).

`.word` *hexconst*{,*hexconst*}

You cannot deposit more than 10 words in a single deposit.

The full deposit entry may be up to 40 characters in length. If you try to type beyond the 40 character limit, the cursor will briefly flash as a ?.

Example deposit entries:

```
.byte "This is a string"

.byte $00,$ff,37

.word 6543,ff,00c0,1a,c
```

Command: **rg**

Super: see **reg** in Chapter 8.

Purpose: open registers for viewing and modification.

Usage: **rg**

Note: takes no parameters.

rg allows the display and modification of all the 6502 registers and the Commodore memory map register. When you open registers with the **rg** command, you are placed in an interactive mode where all keystrokes are intercepted and handled specially. In **rg**-mode you are able to view each register in turn and modify any one at will.

Output for the **rg** command is identical to the super-debugger **reg** command, except that the value in the PC register is not shown in symbolic form.

Mini-debugger Open rg-mode Keystrokes

When data is opened with the **rg** command, the super-debugger is intercepting keystrokes and responding at that level. When an invalid keystroke or a bad entry is detected, the cursor will briefly flash as a ? symbol. The following keystrokes have an effect.

SPACE	enter deposit mode at this location (see deposit description below)
↓ or J	close current register and open next one.
SHIFT + ↓ or K	close current register and open previous one.
RETURN	close current register and return to command prompt.

Deposit rg-mode

When you press space at the asterisk prompt, the data field clears and the cursor is placed into it. At this point you can enter new data for this register. As on the command line, DEL deletes the character to the left of the cursor and ← clears the input line.

To enter a line and leave deposit mode, use one of the following keystrokes:

⇆	enter current line and reopen current instruction.
↕	enter current line and open next register.
SHIFT + ↕	enter current line and open previous register.
RETURN	enter current line and return to command prompt.

If an error is detected in the entry, the line will not be entered and the cursor will briefly flash as a ?.

To leave deposit mode without entering a line, do one of the following:

1. Enter an empty line or a line which contains only spaces.

2. Use DEL to backspace out of the disassembly/deposit field.

rg-mode Deposit Syntax

rg-mode deposits have the following syntax:

hexconst

If the register size is byte, only the low-byte of the expression will be stored in the register.

Command: **fg**

Super: see **flag** in Chapter 8.

Purpose: open individual flags in the processor status register (ST)

Usage: **fg**

Note: takes no parameters.

The **fg** command is identical to the super-debugger **flag** command in all respects except that it does not take a parameter (refer to **flag** in Chapter 8).

Execution Commands

Command: **go**

Mini: see **go** in Chapter 8.

Purpose: Begin full-speed execution of program.

Usage: **go** [***hexconst***]

Note: *hexconst* is the address to begin execution; if no address is given, execution will begin at the current location of the program counter (PC).

The **go** command starts full speed execution of the program. The GEOS screen is displayed and a **jmp** to the proper address is simulated. Control will not return to the mini-debugger unless a breakpoint or a **brk** instruction is encountered or the RESTORE key is pressed.

Examples:

go 450 *begins execution at $450.*

go *begins execution at the program counter.*

Command: **rt**

Super: see **runto** in Chapter 8.

Purpose: execute until a given address is reached.

Usage: **rt *hexconst***

Note: *hexconst* is the address where execution will stop. If the *hexconst* is omitted, the current value of the location counter will be used.

The **rt** command automates the common debugging procedure of setting a breakpoint, performing a **go** to current location of the program counter, and clearing the breakpoint when control returns to the debugger. If no stop address is specified, the current value of the location counter will be used. This allows you to run to the address of the last memory location disassembled.

Example:

rt 251e *sets a breakpoint at **$251e** and executes a **go** to the current location of the program counter.*

Command: **js**

Super: see **jsr** in Chapter 8.

Purpose: Execute a subroutine (**jsr**).

Usage: **js** ***hexconst***

Note: *hexconst* is the address of the subroutine to execute.

The **js** command allows you to execute a subroutine. The mini-debugger will simulate a top-step (see **t** command) through an actual **jsr** instruction. The routine at *hexconst* is expected to return with an **rts**.

Example:

js $680 *executes a* **jsr** *to the routine at* ***$680.*** *Control returns to the mini-debugger when an* **rts** *is encountered.*

Command: **s**

Super: see **s** in Chapter 8.

Purpose: single step through instructions and into subroutines.

Usage: **s**

Note: takes no parameters.

The mini-debugger **s** command is identical to the super-debugger **s** command in all respects except that it will not accept a conditional breakpoint expression (refer to **s** in Chapter 8).

Command: **t**

Super: see **t** in Chapter 8.

Purpose: single-step through instructions and top-step through subroutines.

Usage: **t**

Note: takes no parameters

The mini-debugger **t** command is identical to the super-debugger **t** command in all respects except that it will not accept a conditional breakpoint expression (refer to **t** in Chapter 8).

Command: **nx**

Super: see **next** in Chapter 8.

Purpose: Set breakpoint at next instruction (physically in memory) and proceed with current instruction.

Usage: **nx**

Note: takes no parameters.

The mini-debugger **nx** command is functionally identical to the super-debugger **next** command (refer to **next** in Chapter 8).

Command: **sm**

Super: see **stopmain** in Chapter 8.

Purpose: Continue program execution until a safe point in the GEOS **MainLoop**, then return to the mini-debugger.

Usage: **sm**

Note: takes no parameters.

The mini-debugger **sm** command is functionally identical to the super-debugger **stopmain** command (refer to **stopmain** in Chapter 8).

IMPORTANT: If you break into the debugger with the RESTORE key while interrupt code is being executed and you *do not* do an **sm**, a subsequent **gb** or **pb** could destory a disk.

Breakpoint Commands

The mini-debugger and the super-debugger use the same basic algorithms for managing breakpoints. Refer to "Breakpoint Commands" in Chapter 8 for a complete discussion of breakpoints.

Command: **b**

Super: see **b** in Chapter 8.

Purpose: display currently active breakpoints.

Usage: **b**

Note: takes no parameters

To view the currently active breakpoints, use the **b** command without a parameter. The locations of the currently set breakpoints will be disassembled. For example:

```
0402 8D 34 40     b  sta $4034     first breakpoint
3FCA 20 00 20     b  jsr $2000     second breakpoint
5602 85 F8        b  sta $F8       third breakpoint
```

If no breakpoints are set, no lines will be printed.

Command: **sb**

Super: see **setb** in Chapter 8.

Purpose: set a breakpoint

Usage: **setb** [*hexconst*]

Note: *hexconst* is the address where the breakpoint should be set. If and address is not specified, a breakpoint will be set at the address of the current location counter.

The **sb** command allows you to set a breakpoint in memory. To set a breakpoint at a specific memory location, merely supply a *hexconst* as a parameter. **sb** will set a breakpoint at that location

Example:

sb $4fe *sets a breakpoint at $4fe*

If you use **sb** without a parameter, a breakpoint will be set at the current address of the location counter . The location counter is a value maintained by geoDebugger. It holds the address of the most recently opened or displayed memory location. For example, after a **w** command, the location counter points to the address of the last instruction disassembled. Following a **w** with a **sb** without a parameter would set a breakpoint at this last instruction.

Example:

If the last memory location opened was **$3245**,

sb

would set a breakpoint at this location.

NOTE: It is often easier to set breakpoints with the **a** and **m** open mode commands.

Command: **cb**

Super: see **clrb** in Chapter 8.

Purpose: clear a single breakpoint.

Usage: **cb** [*hexconst*]

Note: *hexconst* is the address of the breakpoint to clear. If an address is not specified, the breakpoint at the current location counter will be cleared.

The **cb** command allows you to clear a breakpoint in memory. To clear a breakpoint at a specific memory location, merely supply a *hexconst* as a parameter. **cb** will clear the breakpoint at that location. If there is no breakpoint at that location, the **cb** command will produce an error.

Example:

clrb $4001 *clears a breakpoint at $4001*

If you use **cb** without a parameter, the breakpoint at the current address of the location counter will be cleared. The location counter is a value maintained by geoDebugger. It holds the address of the most recently opened or displayed memory location. For example, after a **b** command, the location counter points to the address of the last breakpoint disassembled. Following a **b** with a **cb** without a parameter clears the last breakpoint listed.

NOTE: It is often easier to clear breakpoints with the **a** and **m** open mode commands.

Command: **ib**

Super: see **initb** in Chapter 8.

Purpose: initialize (clear) all breakpoints.

Usage: **ib**

Note: takes no parameters

The **ib** command will clear all currently active breakpoints.

Special Commands

Command: **pc**

Super: see **pc** in Chapter 8.

Purpose: view or set program counter.

Usage: **pc** [*hexconst*]

Note: *hexconst* is the address to set the program counter at; the new address of the program counter will be disassembled to the screen. If an address is not specified, the current value of the program counter will be disassembled to the screen.

The **pc** command is a quick and easy way to set the program counter. As a side benefit, the **pc** command (with or without a parameter) will also set the location counter to the address of the program counter, thereby causing a subsequent command which uses that value, such as **m** or **a**, to begin at the program counter.

Disk Commands

Command: **da, db, di**

Super: see **drivea, driveb,** and **disk** in Chapter 8.

Purpose: set current drive or display name of disk in current drive.

Usage: **da**
db
di

Note: takes no parameters.

The **da** and **db** commands open the disk in drive A or drive B, respectively, and make that drive the current drive. Subsequent disk commands will access the current drive. **di** displays the name of the disk in the current drive. These commands call the GEOS **SetDevice** and **OpenDisk** routines.

Command: **gb**

Super: see **getb** in Chapter 8.

Purpose: get block from the disk in the current drive.

Usage: **gb** [*track,sector*]

Note: *track* is a valid track number *hexconst* and *sector* is a valid sector number *hexconst* for the current drive. If the track and sector are not provided, the values in the GEOS **r1L** and **r1H** registers will be used.

The **gb** command reads one sector from the current drive into **diskBlkBuf** at $8000 and then executes a **dd** command to display the sector. The values of the track and sector number read will be left in **r1L** and **r1H**; a subsequent **pb** could then be used to write out the sector just read. **gb** calls the GEOS **GetBlock** routine.

gb 12,0 *get the first block of the directory.*

Command: **pb**

Super: see **putb** in Chapter 8.

Purpose: put a block to the disk in the current drive.

Usage: **pb** [*track,sector*]

Note: *track* is a valid track number *hexconst* and *sector* is a valid sector number *hexconst* for the current drive. If the track and sector are not provided, the values in the GEOS **r1L** and **r1H** registers will be used.

The **pb** command writes one sector from **diskBlkBuf** at $8000 to the disk in the current drive. The values of the track and sector number written will be left in **r1L** and **r1H**. **pb** calls the GEOS **PutBlock** routine.

IMPORTANT: Be careful using **pb**, especially with no parameters; it is very easy to destroy a disk by writing to the wrong track and sector, especially if **r1L** or **r1H** contain bad values.

Example:

pb 5,2 put a block at track $5, sector $2.

HINT: **gb** and **pb** can be used together. You can read in a specific sector with **gb**, modify it in **diskBlkBuf** (without affecting **r1L** and **r1H**) and then write it back out again by using **pb** with no parameters.

Command: **dd**

Super: see **dumpd** in Chapter 8.

Purpose: dump disk block buffer (**diskBlkBuf**)

Usage: **dd**

Note: takes no parameters.

The **dd** command dumps all 256 bytes of the disk block buffer (**diskBlkBuf**) at address $8000 in the standard **d** command.

Appendix A: Library Files and Sample Source

Your geoProgrammer disk contains a number of geoAssembler source code files (equates, macros, and sample applications) for you to use as the basis of your own projects.

GEOS Equates and Constants Files

Any GEOS application will a spend a great deal of its time calling routines within the GEOS Kernal. Each GEOS routine has a name, and when you call GEOS routines using these names, you make your source code more readable as well as consistent with other GEOS source code and the Berkeley Softworks design methodology. In addition to routine addresses, GEOS also uses a large number of constants (for selecting colors and object attributes) and global variables, all of which also have names. The names for these routines, constants, and variables were first published in *The Official GEOS Programmer's Reference Guide*, where they are described in detail. On your geoProgrammer disk are four files which you can include in your assemblies (using the **.include** directive):

With comments:
geosConstants
geosMemoryMap
geosRoutines

Combined and without comments:
geosSym

These files contain the same equates and constants which are described in the *GEOS Programmer's Reference Guide*, except that some names have been changed so that they differ from all other GEOS names in the first eight characters. Equate and constant names which have been changed have an asterisk at the last tab stop in the comment field (only in the commented versions, though). Some unnecessary equates and constants have also been removed, while others have been equated with the (=) directive (as opposed to ==) to avoid sending them to the debugger. For a more complete discussion of equates involved, refer to the the listings in the *GEOS Programmer's Reference Guide.*

NOTE: The uncommented and combined version of the include files (**geosSym**) have been compacted by removing spaces and comments, reducing its size by two thirds; the commented and uncommented versions are otherwise identical. The commented versions of the files are for reference, while the uncommented version, which take up less disk space and assembles more quickly, is for including in your programs.

Macro Files

Also included on your geoProgrammer disk is a file of useful macros:

geosMacros	(with comments)
geosMac	(without comments)

The macros in these files were chosen based on their utility and their ability to demonstrate the flexibility of the macro processor. They are not intended to be a comprehensive set: there are many variations on these basic macros which you can build as you develop the need. The primary limitation is the size of the macro buffer.

Summary of geosMacros File

There are 29 macros defined in the **geosMacros** (**geosMac** without comments) file, and each one has a specific use. Below is a brief discussion of each. (Register notation: A = Accumulator, ST = Status register, SP = Stack Pointer; all macros are assumed to affect the Program Counter.)

Load and Move

Load Byte **LoadB** ***dest, value***

Loads a memory address (*dest*) with an immediate byte (*value*). Affects the A and ST registers.

Load Word **LoadW** ***dest, value***

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. Affects the A and ST registers.

Move Byte **MoveB** ***source, dest***

Moves a byte from one address (*source*) to another address (*dest*). The byte at the source address is not destroyed. Affects the A and ST registers.

Move Word **MoveW** ***source, dest***

Moves a word (two bytes) from one address (*source*) to another address (*dest*). The word at the source address is not destroyed. Affects the A and ST registers.

Addition

Add Byte **add** ***addend***

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. Affects the A and ST registers.

Add Bytes **AddB** ***source, dest***

Adds the byte at one address (*source*) to the byte at another address (*dest*) and stores the result in *dest*. Affects the A and ST registers.

Add Words **AddW** ***source, dest***

Adds the low, high word at *source* and *source*+1 to the word at *dest* and *dest*+1 and stores the result in *dest*. Affects the A and ST registers..

Add Value to Byte **AddVB** ***value, dest***

Adds an immediate byte value to the byte at *dest* and stores the result in *dest*. Affects the A and ST registers.

Add Value to Word **AddVW** ***value, dest***

Adds an immediate byte or word value to the low, high word at *dest* and *dest*+1 and stores the result in *dest*. Affects the A and ST registers..

Subtraction

Subtract Byte **sub** ***subtrahend***

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 A-register. If it is an immediate value (preceded by a # sign), the actual value is subtracted from the A register. The result is returned in the A-register. The sole purpose of the **sub** macro is to combine the **sbc** with its mandatory sec instruction. Affects the A and ST registers.

Subtract Bytes **SubB** ***source, dest***

Subtracts the byte at one address (*source*) from the byte at another address (*dest*) and stores the result in *dest*. Affects the A and ST registers.

Subtract Words **SubW** ***source, dest***

Subtracts the low, high word at *source* and *source*+1 from the word at *dest* and *dest*+1 and stores the result in *dest*. Affects the A and ST registers.

Subtract Value from Byte **SubVB** ***value, dest***

Subtracts an immediate byte value from the byte at *dest* and stores the result in *dest*. Affects the A and ST registers.

Subtract Value from Word **SubVW** ***value, dest***

Subtracts an immediate byte or word value from the low, high word at *dest* and *dest*+1 and stores the result in *dest*. Affects the A and ST registers.

Comparison

Compare Bytes **CmpB** ***source, dest***

Compares the byte at *source* to the byte at *dest*. Affects the A and ST registers.

Compare Byte to Value **CmpBI** ***source, value***

Compares the byte at *source* with the immediate byte *value*. Affects the A and ST registers.

Compare Words **CmpW** ***source, dest***

Compares the low, high word at *source* and *source*+1 with the low, high word at *dest* and *dest*+1. Note: the high-bytes are compared first, so the condition codes (and therefore subsequent branches) are the same as for one-byte comparisons. Affects the A and ST registers.

Compare Word to Value **CmpWI** ***source, value***

Compares the word value at *source* and *source*+1 to the immediate word *value*. As with **CmpW**, the condition codes (and therefore subsequent branches) are the same as for one-byte comparisons. Affects the A and ST registers.

Stack Operations

Push Byte **PushB** ***source***

Pushes the byte at *source* onto the stack. *source* can be an immediate value preceded by a # -sign if desired. Affects the A, ST, and SP registers.

Push Word **PushW** ***source***

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*. Affects the A, ST, and SP registers.

Pop Byte **PopB** ***dest***

The opposite of **PushB**; pops a byte from the stack and stores it at *dest*. Affects the A, ST, and SP registers.

Pop Word **PopW** ***dest***

The opposite of **PushW**; pops a word (two-bytes) from the stack and stores it at *dest* and *dest*+1. The first byte popped is the low-byte and is stored and *dest*; the second byte is the high-byte and is stored at *dest*+1. Affects the A, ST, and SP registers.

Unconditional Branch

Branch Relative Always **bra** ***addr***

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. Affects the ST register.

Bit Operations

Set Bit **smb** ***bitNumber, dest***

Sets 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. Affects the ST register.

Set Bit Fast **smbf** ***bitNumber, dest***

Identical to **smb**, except that it is faster and it affects the A and ST registers.

Reset Bit **rmb** ***bitNumber, dest***

Resets (clears) 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. Affects the ST register.

Reset Bit Fast **rmbf** ***bitNumber, dest***

Identical to **rmb**, except that it is faster and it affects the A and ST registers.

Bit Test and Branch Operations

Branch on Bit Set **bbs** ***bitNumber, source, addr***

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.

Branch on Bit Set Fast **bbsf** ***bitNumber, source, addr***

Identical to **bbs**, except it is faster and affects the A and ST registers.

Branch on Bit Reset **bbr** ***bitNumber, source, addr***

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 (cleared), a relative branch to *addr* is taken. Otherwise, it falls through to the next instruction. Does not affect *any* registers.

Branch on Bit Reset Fast **bbrf** ***bitNumber, source, addr***

Identical to **bbs**, except it is faster and affects the A and ST registers.

(For more information on these macros, refer to the commented source file **geosMacros**.)

Sample Applications

Your geoProgrammer disk contains three sample applications in geoAssembler source code format. There is one sequential application, one VLIR application (with overlay modules), and one desk accessory. Each has its own source code modules, header definitions, and link command files. Everything you need to assemble, link, and execute these applictions is included on your geoProgrammer disk. (For more information on assembling and linking the sample sequential application, refer to ."Creating a Sample Application" in Chapter 4.)

The sample applications serve to demonstrate the usage of geoAssembler and geoLinker. They also show successful coding conventions, such as modular design and commenting, as well as offering functional GEOS source code, supplementing the *GEOS Programmer's Reference Guide*. But perhaps their most useful purpose will be as a base for your own applications. Although the code is copyrighted by Berkeley Softworks, free license is granted for all registered owners of geoProgrammer to use the source code in their own applications, modifying and adding to it as they see fit. In fact, Berkeley Softworks encourages you to use the sample files as a shell for your applications, as they do in their cross-development environment.

Sample Sequential Application

The sample sequential application consists of one main module. This module contains all the initialization code, event handling code, as well as the object structures for a menu and an icon. There is no desk accessory support. Refer to the sample VLIR application for desk accessory management code.

Sample sequential files:

SamSeq	main module
SamSeqHdr	file header
SamSeq.lnk	link command file

In addition, there is also a debugger macro file (**SamSeq.dbm**) for use with the sample application.

Sample Desk Accessory

The sample desk accessory is very similar to the sample sequential application. It, too, consists of one main module which contains all the initialization code, event handling code, and object structures. The sample desk accessory, however, conforms to the GEOS desk accessory protocol, which allows it to operate without corrupting the parent application. Also of interest is the desk accessory header file, which differs from other file headers in a few significant areas.

Sample desk accessory files:

SamDA	main module
SamDAHdr	file header
SamDA.lnk	link command file

Sample VLIR application

The sample VLIR application is the most sophisticated of the sample applications — not only does it use menus and icons, but it has an overlay manager which handles swapping overlay modules in and out of memory as they are needed, as well as the using jump tables to access routines within the modules. Also of interest is the desk accessory support code.

Sample VLIR files:

SamVlirRes	resident module: initialization code, overlay manager, and object structures
SamVlirEdit	overlay module for **Edit** sub-menu
SamVlirFile	overlay module for **File** sub-menu; includes desk accessory management code
SamVlirEquates	internal equate include file
SamVlirZP	internal zero page zsect definitions
SamVlirHdr	VLIR file header
SamVlir.lnk	link command file

Sample VLIR Application Roadmap

Because of the complexity of the sample VLIR application, the following outline, or "roadmap," will help you decipher its inner-workings.

Initialization

When **SamVlir** is opened from the deskTop, the resident module is loaded into memory and executed. At this point, none of the overlay modules have been loaded. The first routine executed, **ResStart**, does the following:

1. Clears the screen.
2. Initializes a swapping table to facilitate overlay management.
3. Initializes the menu structure, placing the appropriate desk accessory entries under the **Geos** menu.
4. Initializes the icon structure.
5. Jumps to the GEOS **Mainloop**. GEOS **Mainloop** now runs continuously, waiting for events. When an event occurs, such as a when a menu item is selected or an icon is clicked on, **Mainloop** jumps through the appropriate event vectors to our code.

Below are the events which the sample VLIR application recognizes and handles:

Geos menu

When **SampleVlir info** is selected, **Mainloop** calls our event routine **R_DoAbout**. **R_DoAbout** is an empty routine which simply returns with an **rts**.

When a desk accessory is selected, **Mainloop** calls our event routine, **R_RunDA**, which does the following:

1. Calls **SwapMod**, which will load the **SampleVlirFile** overlay module if it is not already in memory.
2. Calls **RunDA** through the jump table equate **J_RunDA**.
3. **RunDA** does the following:
   A. Saves some miscellaneous system status items to restore later.
   B. Uses **GetFile** to load the desk accessory and save out the area of memory which the desk accessory overlays.
   C. Passes control to the desk accessory and awaits its return.
   D. The desk accessory returns, and the system status (including the overlayed memory) is restored.
   E. Returns through resident **R_RunDA** routine.

R_RunDA now returns back to the GEOS **mainloop**, waiting for another event.

File menu

close When this menu item is selected, GEOS calls the resident routine **R_DoClose**, which does the following:

1. Calls **SwapMod** to load the **SampleVlirFile** module.
2. Calls **DoClose** through the jump table equate **J_DoClose**. **DoClose** is an empty routine and merely returns to the resident **R_DoClose**.
3. Returns to the GEOS **Mainloop**.

Edit menu

cut When this menu item is selected, GEOS calls the resident routine **R_DoCut**, which does the following:

1. Calls **SwapMod** to load the **SampleVlirEdit** overlay module.
2. Calls **DoCut** through the jump table equate **J_DoCut**. **DoCut** is an empty routine and merely returns to the resident **R_DoCut**.
3. Returns to the GEOS **Mainloop**.

copy Similar to **cut**.

paste Similar to **cut**.

Icon press

When the icon is clicked on, GEOS **Mainloop** calls **R_DoIcon1**, which does the following:

1. Calls **SwapMod** to load the **SampleVlirEdit** overlay module.
2. Calls **DoIcon1** through the jump table equate **J_DoIcon1**. **DoIcon1** is an empty routine and merely returns to the resident **R_DoIcon1**.
3. Returns to the GEOS **Mainloop**.

```
;*************************************************************************
;
;                                   geosConstants
;
;This file contains equates for use in GEOS applications.
;
;Copyright (c) 1987 Berkeley Softworks. For the sole use of registered
;GeoProgrammer owners.
;*************************************************************************

;*************************************************************************
;                       Miscellaneous Equates
;*************************************************************************

TRUE                    = -1
FALSE                   = 0

;*************************************************************************
;                       Hardware Related Equates
;*************************************************************************

;The following equates define the numbers written to the CPU_DATA register
;(location $0001 in C-64). These numbers control the hardware memory map
;of the C-64.

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

;*************************************************************************
;                       Menu Equates
;*************************************************************************

;Menu types

HORIZONTAL              = %00000000
VERTICAL                = %10000000
CONSTRAINED             = %01000000
UN_CONSTRAINED          = %00000000

;Offsets to variables in the menu structure

OFF_MY_TOP              = 0         ;offset to y pos of top of menu                 *
OFF_MY_BOT              = 1         ;offset to y pos of bottom of menu              *
OFF_MX_LEFT             = 2         ;offset to x pos of left side of menu           *
OFF_MX_RIGHT            = 4         ;offset to x pos of right side of menu          *
OFF_NUM_M_ITEMS         = 6         ;offset to AlignmentlMovementlNumber of items
OFF_1ST_M_ITEM          = 7         ;offset to record for 1st menu item in structure
```

```
;Types of menu 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

;*************************************************************************
;                       Process Related Equates
;*************************************************************************

;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         ;runable flag
BLOCKED_BIT     = 6         ;process blocked flag
FROZEN_BIT      = 5         ;process frozen flag
NOTIMER_BIT     = 4         ;not a timed process flag

;*************************************************************************
;                       Text Equates
;*************************************************************************

;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 byte as 1+x cursor
GOTOY           = 21    ;use next byte as 1+y cursor
GOTOXY          = 22    ;use next bytes as 1+x and 1+y cursor
NEWCARDSET      = 23    ;use next two bytes as new font id
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
SHORTCUT        = 128   ;shortcut character

;*************************************************************************
;                       Keyboard Equates
;*************************************************************************

;Values for keys

KEY_INVALID     = 31
KEY_F1          = 1
KEY_F2          = 2
KEY_F3          = 3
KEY_F4          = 4
KEY_F5          = 5
KEY_F6          = 6
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_LEFT        = BACKSPACE
KEY_RIGHT       = 30
KEY_DELETE      = 29
KEY_INSERT      = 28

;*************************************************************************
;                       Mouse Equates
;*************************************************************************

;Bit flags for mouseOn variable

SET_MSE_ON      = %10000000 ;                                           *
SET_MENUON      = %01000000
SET_ICONSON     = %00100000

MOUSEON_BIT     = 7
MENUON_BIT      = 6
ICONSON_BIT     = 5

;*************************************************************************
;                       Graphics/Screen Equates
;*************************************************************************

;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 displayBufferOn 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       ;graphics strings only write to foreground.

;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 signed word delta in x & y
```

```
;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

;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

;*************************************************************************
;                       Menu Equates
;*************************************************************************

;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_FLASH        = $80       ;bit to indicate icon should flash
ST_INVERT       = $40       ;bit to indicate icon should be inverted

;offsets into the icon structure

OFF_NM_ICNS     = 0         ;number of icons in structure      *
OFF_IC_XMOUSE   = 1         ;mouse x start position            *
OFF_IC_YMOUSE   = 3         ;mouse y start position            *

;Offsets into an icon record in icon structure.

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       *
```

```
;*****************************************************************************
;                          Flag Equates
;*****************************************************************************

;Values for pressFlag variable

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

;Values for faultFlag variable

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

;*****************************************************************************
;                          GEOS File Type Equates
;*****************************************************************************

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

NOT_GEOS            = 0             ;Old C-64 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             ;C-64 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     ;C-64 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 files)
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         ;CBM BAM protection file, currently only on
                            ;1581 disk drivers.  Used to protect specific
                            ;blocks/tracks from collection at validation
                            ;time.

TOTAL_BLOCKS    = 664       ;number of blocks on disk, not including
                            ;       directory track.

;*************************************************************************
;                       Directory Header Equates
;*************************************************************************

;Offsets into a directory header structure

OFF_TO_BAM      = 4         ;first BAM entry
OFF_DISK_NAME   = 144       ;disk name string
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. Currently, is 0 for            *
                            ;normal disk, 'B' for BOOT disk. Zeroed
                            ;on destination disk during disk copy.

;*************************************************************************
;                       Directory Entry Equates
;*************************************************************************

ST_WR_PR        = $40       ;write protect bit: bit 6 of byte 0 in the
                            ;directory entry

;Offsets within a specific file's Directory Entry.

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

;Offsets into a directory block

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

```
;*************************************************************************
;                          File Header Equates
;*************************************************************************

;offsets into a GEOS file header block

O_GHIC_WIDTH      = 2       ;byte: width in bytes of file icon            *
O_GHIC_HEIGHT     = 3       ;byte: indicates height of file icon          *
O_GHIC_PIC        = 4       ;64 bytes: picture data for file icon         *
O_GHCMDR_TYPE     = 68      ;byte: Comm. file type                        *
O_GHGEOS_TYPE     = 69      ;byte: GEOS file type                         *
O_GHSTR_TYPE      = 70      ;byte: GEOS file structure type               *
O_GHST_ADDR       = 71      ;2 bytes: start address of file in mem        *
O_GHEND_ADDR      = 73      ;2 bytes: end address of file in memory       *
O_GHST_VEC        = 75      ;2 bytes: init vector if file is appl.        *
O_GHFNAME         = 77      ;20 bytes,permanent filename.

O_128_FLAGS       = 96      ;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.

O_GH_AUTHOR       = 97      ;20 bytes: author's name (only if is applic.)
O_GHINFO_TXT      = $A0     ;offset to notes that are stored with the file
                            ;and edited in the deskTop "get info" box.

;if file is an application data file:

O_GHP_DISK        = 97      ;20 bytes: disk name of parent application's
                            ;disk.

O_GHP_FNAME       = 117     ;20 bytes: permanent filename of parent
                            ;application.

;*************************************************************************
;                          GetFile Equates
;*************************************************************************

;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 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.

;*************************************************************************
;                       Disk Equates
;*************************************************************************

DK_NM_ID_LEN    = 18        ; # of characters in disk name

;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"

;Constants used by 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        ;track # reserved on disk for directory
                            ;on a 1581

;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 Error Equates
;*************************************************************************

;The following equates are ERROR values returned from direct access routines

NO_BLOCKS       = 1     ;"not enough blocks"                            *
INV_TRACK       = 2     ;"invalid track"                                *
INSUFF_SPACE    = 3     ;"not enough blocks on disk"                    *
FULL_DIRECTORY  = 4     ;"directory full"
FILE_NOT_FOUND  = 5     ;"file not found"
BAD_BAM         = 6     ;"bad Block Availability Map"
UNOPENED_VLIR   = 7     ;"unopened VLIR file"                           *
INV_RECORD      = 8     ;"invalid record"                               *
OUT_OF_RECORDS  = 9     ;"cannot insert/append more records"
STRUCT_MISMAT   = 10    ;"file structure mismatch"                      *
BFR_OVERFLOW    = 11    ;"buffer overflow during load"                  *
CANCEL_ERR      = 12    ;"deliberate cancel error"
DEV_NOT_FOUND   = 13    ;"device not found"                             *
INCOMPATIBLE    = 14    ;This error is returned when an attempt is made
                        ;to load a program that can't be run on the
                        ;current graphics modes under the C-128 GEOS.
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"

;*************************************************************************
;                       Dialog Box Equates
;*************************************************************************

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

;Dialogue box descriptor table commands

OK              = 1     ;Put up system icon for "OK", command is
                        ;followed by 2 byte position indicator, x pos.
                        ;in bytes, y pos. in pixels. NOTE: positions
                        ;are offsets from the top left corner of the
                        ;dialogue 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

;More dialogue box descriptor table commands

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 applic.
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.

;The following equates are used to specify offsets into a dialogue box
;descriptor table.

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

;The following equates specify the dimensions of the system defined dialogue
;box icons.

SYSDBI_WIDTH    = 6     ;width in bytes                                 *
SYSDBI_HEIGHT   = 16    ;height in pixels                               *

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

DEF_DB_TOP      = 32    ;top y coordinate of default box
DEF_DB_BOT      = 127   ;bottom y coordinate of default box
DEF_DB_LEFT     = 64    ;left edge of default box
DEF_DB_RIGHT    = 255   ;right edge of default box

TXT_LN_X        = 16    ;standard text x start

TXT_LN_1_Y      = 16    ;standard text line y offsets
TXT_LN_2_Y      = 32
TXT_LN_3_Y      = 48
TXT_LN_4_Y      = 64
TXT_LN_5_Y      = 80
```

```
;byte offsets to...

DBI_X_0         = 1         ;left side standard icon x position          *
DBI_X_1         = 9         ;center standard icon x position             *
DBI_X_2         = 17        ;right side standard icon x position         *
DBI_Y_0         = 8         ;left side standard icon y position          *
DBI_Y_1         = 40        ;center standard icon y position             *
DBI_Y_2         = 72        ;right side standard icon y position         *

;*****************************************************************************
;                       VIC Chip Equates
;*****************************************************************************

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.

;Desk Accessory save foreground bit.

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

```
;*****************************************************************************
;
;                                   geosMemoryMap
;
;This file contains equates for use in GEOS applications.
;
;Copyright (c) 1987 Berkeley Softworks. For the sole use of registered
;GeoProgrammer owners.
;*****************************************************************************

;*****************************************************************************
;                        Principal Memory Map Equates
;*****************************************************************************

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
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
cia1base        == $DC00    ;1st communications interface adaptor (CIA).
cia2base        == $DD00    ;second CIA chip
EXP_BASE        == $DF00    ;Base address of RAM expansion unit
MOUSE_JMP       == $FE80    ;start of mouse jump table
MOUSE_BASE      == $FE80    ;start of input driver
END_MOUSE       == $FFFA    ;end of input driver
```

```
;*****************************************************************************
;                          Zero Page Equates and Space Definitions
;*****************************************************************************

CPU_DDR             == $0000    ;address of 6510 data direction register
CPU_DATA            == $0001    ;address of 6510 data register
STATUS              == $0090    ;c64 status register
curDevice           == $00BA    ;current serial device #

;zero page variable definitions:

zpage               = $0000     ;6510 registers: CPU_DDR and CPU_DATA
r0                  == $0002
r1                  == $0004
r2                  == $0006
r3                  == $0008
r4                  == $000a
r5                  == $000c
r6                  == $000e
r7                  == $0010
r8                  == $0012
r9                  == $0014
r10                 == $0016
r11                 == $0018
r12                 == $001a
r13                 == $001c
r14                 == $001e
r15                 == $0020
```

```
;The following variables are saved by GEOS during dialog boxes and desk
;accessories.

curPattern          == $0022    ;pointer to the current pattern                 *
string              == $0024
baselineOffset      == $0026    ;Offset from top line to baseline in
                                ;character set
curSetWidth         == $0027    ;Card width in pixels                           *
curHeight           == $0029    ;Card height in pixels                          *
curIndexTable       == $002a    ;Size of each card in bytes                     *
cardDataPntr        == $002c    ;Pointer to the actual card                     *
                                ;graphics data
currentMode         == $002e    ;Current underline, italic and reverse flags
dispBufferOn        == $002f    ;bit 7 controls writes to FG screen             *
                                ;bit 6 controls writes to background screen
mouseOn             == $0030    ;flag indicating that the mouse mode is on
msePicPtr           == $0031    ;pointer to mouse graphics data                 *
windowTop           == $0033    ;top line of window for text clipping
windowBottom        == $0034    ;bottom line of window for text clipping
leftMargin          == $0035    ;leftmost point for writing characters.
                                ;CR will return to this point
rightMargin         == $0037    ;rightmost point for writing characters. When
                                ;crossed, call mode through StringFaultVector

;End of variables saved during DB's and DA's.

pressFlag           == $0039    ;Flag indicating that a new key has been pressed
mouseXPos           == $003a    ;x position of mouse                            *
mouseYPos           == $003c    ;y position of mouse                            *
returnAddress       == $003d    ;address to return from in-line call

;equates to access low and high bytes of general purpose registers:

r0L                 == $02
r0H                 == $03
r1L                 == $04
r1H                 == $05
r2L                 == $06
r2H                 == $07
r3L                 == $08
r3H                 == $09
r4L                 == $0a
r4H                 == $0b
r5L                 == $0c
r5H                 == $0d
r6L                 == $0e
r6H                 == $0f
r7L                 == $10
r7H                 == $11
r8L                 == $12
r8H                 == $13
r9L                 == $14
r9H                 == $15
```

```
r10L            == $16
r10H            == $17
r11L            == $18
r11H            == $19
r12L            == $1a
r12H            == $1b
r13L            == $1c
r13H            == $1d
r14L            == $1e
r14H            == $1f
r15L            == $20
r15H            == $21

;Zero Page variables for use by applications ONLY! Not to be used by
;GEOS or desk accessories.

a0              == $fb
a0L             == $fb
a0H             == $fc
a1              == $fd
a1L             == $fd
a1H             == $fe
a2              == $70      ;Notice jump here to lower memory
a2L             == $70
a2H             == $71
a3              == $72
a3L             == $72
a3H             == $73
a4              == $74
a4L             == $74
a4H             == $75
a5              == $76
a5L             == $76
a5H             == $77
a6              == $78
a6L             == $78
a6H             == $79
a7              == $7a
a7L             == $7a
a7H             == $7b
a8              == $7c
a8L             == $7c
a8H             == $7d
a9              == $7e
a9L             == $7e
a9H             == $7f
```

```
;*************************************************************************
;                       $0300 Area Equates and Space Definitions
;*************************************************************************

irqvec          == $0314    ;irq vector (two bytes)
bkvec           == $0316    ;break ins vector (two bytes)
nmivec          == $0318    ;nmi vector (two bytes)
kernalVectors   == $031A    ;location of kernal vectors

;*************************************************************************
;                       $8000 Area Equates and Space Definitions
;*************************************************************************

;Start of GEOS system RAM

diskBlkBuf      == $8000    ;general disk block buffer
fileHeader      == $8100    ;block used to hold the header block for a
                            ;GEOS file.
curDirHead      == $8200    ;block contains directory header information for
                            ;disk in currently selected drive.
fileTrScTab     == $8300    ;buffer used to hold track and sector chain for
                            ;a file of maximum size 32,258 bytes.
dirEntryBuf     == $8400    ;buffer used to build a files directory entry

;Disk variables

DrACurDkNm      == $841e    ;Disk name of disk in drive A
                            ;18 char disk name (padded with $A0)
DrBCurDkNm      == $8430    ;Disk name of disk in drive B
                            ;18 char disk name (padded with $A0)
dataFileName    == $8442    ;Name of data file (passed to application)
dataDiskName    == $8453    ;Disk that data file is on.
PrntFilename    == $8465    ;Name of current printer driver
                            ;16 byte filename, 1 byte terminator
PrntDiskName    == $8476    ;Disk that current printer driver resides on
                            ;disk name plus terminator byte

curDrive        == $8489    ;currently active disk drive (8,9,10 or 11)
diskOpenFlg     == $848a    ;indicates if a disk is currently open
isGEOS          == $848b    ;flag indicates if current disk is a GEOS disk
interleave      == $848c    ;BlkAlloc uses the value here as the desired
                            ;interleave when selecting free blocks.
numDrives       == $848d    ; # of drives running on system.
driveType       == $848e    ;Disk Drive types: 1 byte drive type for each
                            ;of drives 8,9,10,11.
turboFlags      == $8492    ;Turbo state flags for drives 8, 9, 10, and 11.
```

```
;Variables kept current for a specific opened file of structure type VLIR

curRecord       == $8496    ;current record #
usedRecords     == $8497    ;number of records in open file
fileWritten     == $8498    ;flag indicating if file has been written to
                            ;since last update of index Tab & BAM
fileSize        == $8499    ;current size (in blocks) of file. This is
                            ;pulled in from & written to directory entry.

;The following variables are saved by GEOS during dialog boxes and desk
;accessories.

;Vectors

appMain         == $849b    ;Application's main loop code. Allows          *
                            ;apps to include their own main loop at the
                            ;end of OS main loop
intTopVector    == $849d    ;Called at the top of OS interrupt code        *
                            ;to allow application programs to have interrupt
                            ;level routines.
intBotVector    == $849f    ;Called at the bottom of OS interrupt          *
                            ;code to allow application programs to have
                            ;interrupt level routines
mouseVector     == $84a1    ;routine to call on mouse key press
keyVector       == $84a3    ;routine to call on keypress
inputVector     == $84a5    ;routine to call on input device change
mouseFaultVec   == $84a7    ;routine to call when mouse goes               *
                            ;outside region or off a menu
otherPressVec   == $84a9    ;routine to call on mouse press that           *
                            ;is not a menu or an icon
StringFaultVec  == $84ab    ;vector for when character written             *
                            ;over rightMargin
alarmTmtVector  == $84ad    ;address of a service routine for the alarm
                            ;clock time-out (ringing, graphic etc.) that
                            ;the App. can use if necessary. Normally 0.

BRKVector       == $84af    ;routine called when BRK encountered
RecoverVector   == $84b1    ;routine called to recover background behind
                            ;menus and dialogue boxes
selectionFlash  == $84b3    ;speed at which menu items and icons are flashed
alphaFlag       == $84b4    ;flag for alphanumeric input
iconSelFlag     == $84b5    ;indicates how to flash icons when selected
faultData       == $84b6    ;Bit flags for mouse faults
menuNumber      == $84b7    ;number of currently working menu
mouseTop        == $84b8    ;top most position for mouse
mouseBottom     == $84b9    ;bottom most position for mouse
mouseLeft       == $84ba    ;left most position for mouse
mouseRight      == $84bc    ;right most position for mouse
```

```
;Global variables for string input and prompt manipulation

stringX          == $84be    ;X position for string input
stringY          == $84c0    ;Y position for string input

;End of variables saved during DB's and DA's.

mousePicData     == $84c1    ;ram array for mouse picture data.
maxMouseSpeed    == $8501    ;maximum speed for mouse                    *
minMouseSpeed    == $8502    ;minimum speed for mouse                    *
mouseAccel       == $8503    ;acceleration of mouse                      *
keyData          == $8504    ;This is where key service routines should look
mouseData        == $8505    ;This is where mouse service routines
                             ;should look
inputData        == $8506    ;This is where input drivers pass device
                             ;specific info to applications that want it
random           == $850a    ;random number, incremented each interrupt
saveFontTab      == $850c    ;when going into menus, save user active font
                             ;table here
dblClickCount    == $8515    ;used to determine double clicks on icons.
year             == $8516
month            == $8517
day              == $8518
hour             == $8519
minutes          == $851a
seconds          == $851b
alarmSetFlag     == $851c    ;TRUE if the alarm is set for geos to monitor.

;dialog box variables

sysDBData        == $851d    ;used internally to indicate which command
                             ;caused a return to the application
                             ;(in dialogue boxes). Actual data is
                             ;returned in r0L.
screencolors     == $851e    ;default screen colors
dlgBoxRamBuf     == $851f    ;buffer to hold variables while DB or DA
                             ;is running

;Second global memory area:

savedmoby2       == $88bb    ;Saved value of moby2 for context saving done
                             ;in dlg boxes & desk accessories. Left out
                             ;of original GEOS save code, put here so we
                             ;don't screw up desk accessories, etc. that
                             ;know the size of TOT_SRAM_SAVED above.
scr80polar       == $88bc    ;Copy of reg 24 in VDC for C128             *
scr80colors      == $88bd    ;Screen colors for 80 column                *
                             ;mode on C128. Copy of reg 26 in VDC.
vdcClrMode       == $88be    ;Holds current color mode for C128 color rtns.
driveData        == $88bf    ;1 byte each reserved for disk drivers
                             ;about each device (each driver may use
                             ;differently).
```

```
ramExpSize          == $88c3
sysRAMFlg           == $88c4    ;If RAM expansion in, Bank 0 is
                                ;reserved for the kernal's use.  This
                                ;byte contains flags designating its
                                ;usage:
                                ;Bit 7:  if 1, $0000-$78FF used by
                                ;MoveData routine
                                ;Bit 6:  if 1, $8300-$B8FF holds disk drivers
                                ;for drives A through C
                                ;Bit 5:  if 1, $7900-$7DFF is loaded with GEOS
                                ;ram area $8400-$88FF by ToBasic routine when
                                ;going to BASIC.
                                ;Bit 4:  if 1, $7E00-$82FF is loaded with reboot
                                ;code by a setup AUTO-EXEC file, which is loaded
                                ;by the restart code in GEOS at $C000 if this
                                ;flag is set, at $6000, instead of loading
                                ;GEOS_BOOT.  Also, in the area $B900-$FC3F is
                                ;saved 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).

firstBoot           == $88c5    ;This flag is changed from 0 to $FF after
                                ;deskTop comes up for the first time
                                ;after booting.
curType             == $88c6    ;Current disk type (copied from diskType)
ramBase             == $88c7    ;RAM bank for each disk drive to use
                                ;if drive type is RAM DISK or Shadowed Drive.
inputDevName        == $88cb    ;Holds name of current input device.
DrCCurDkNm          == $88dc    ;Disk name of disk in drive C
                                ;18 char disk name (padded with $A0)
DrDCurDkNm          == $88ee    ;Disk name of disk in drive D
                                ;18 char disk name (padded with $A0)

dir2Head            == $8900    ;2nd directory header block, for larger disk
                                ;capacity drives (such as 1571)

;Addresses of specific sprite picture data

spr0pic             == $8a00
spr1pic             == $8a40
spr2pic             == $8a80
spr3pic             == $8ac0
spr4pic             == $8b00
spr5pic             == $8b40
spr6pic             == $8b80
spr7pic             == $8bc0
```

```
;Addresses of pointers to sprite object graphics

obj0Pointer             == $8ff8
obj1Pointer             == $8ff9
obj2Pointer             == $8ffa
obj3Pointer             == $8ffb
obj4Pointer             == $8ffc
obj5Pointer             == $8ffd
obj6Pointer             == $8ffe
obj7Pointer             == $8fff

;*************************************************************************
;                       $c000 Area Equates and Space Definitions
;*************************************************************************

bootName                == $C006    ;start of "GEOS BOOT" string
version                 == $C00F    ;GEOS version byte
nationality             == $C010    ;nationality byte
sysFlgCopy              == $C012    ;copy of sysRAMFlg saved here when
                                    ;going to BASIC
dateCopy                == $C018    ;copy of year, month, day

;*************************************************************************
;                       $d000 area: VIC II graphics chip definitions and equates
;*************************************************************************

mob0xpos                == $d000
mob0ypos                == $d001
mob1xpos                == $d002
mob1ypos                == $d003
mob2xpos                == $d004
mob2ypos                == $d005
mob3xpos                == $d006
mob3ypos                == $d007
mob4xpos                == $d008
mob4ypos                == $d009
mob5xpos                == $d00a
mob5ypos                == $d00b
mob6xpos                == $d00c
mob6ypos                == $d00d
mob7xpos                == $d00e
mob7ypos                == $d00f
msbxpos                 == $d010
grcntrl1                == $d011    ;graphics control register #1
rasreg                  == $d012    ;raster register
lpxpos                  == $d013    ;light pen x position
lpypos                  == $d014    ;light pen y position
mobenble                == $d015    ;moving object enable bits.
grcntrl2                == $d016    ;graphics control register #2
moby2                   == $d017    ;double object size in y
```

```
grmemptr        == $d018    ;graphics memory pointer VM13-VM10|CB13-CB11
grirq           == $d019    ;graphics chip interupt register.
grirqen         == $d01a    ;graphics chip interupt enable register.
mobprior        == $d01b    ;moving object to background priority
mobmcm          == $d01c    ;moving object multi-color mode select.
mobx2           == $d01d    ;double object size in x
mobmobcol       == $d01e    ;object to object collision register.
mobbakcol       == $d01f    ;object to background collision register.
extclr          == $d020    ;exterior(border) color.
bakclr0         == $d021    ;background #0 color
bakclr1         == $d022    ;background #1 color
bakclr2         == $d023    ;background #2 color
bakclr3         == $d024    ;background #3 color
mcmclr0         == $d025    ;object multi-color mode color 0
mcmclr1         == $d026    ;object multi-color mode color 1
mob0clr         == $d027    ;object color
mob1clr         == $d028    ;object color
mob2clr         == $d029    ;object color
mob3clr         == $d02a    ;object color
mob4clr         == $d02b    ;object color
mob5clr         == $d02c    ;object color
mob6clr         == $d02d    ;object color
mob7clr         == $d02e    ;object color

;*************************************************************************
;               $f000 Area Equates
;*************************************************************************

NMI_VECTOR      == $fffa    ;nmi vector location
RESET_VECTOR    == $fffc    ;reset vector location
IRQ_VECTOR      == $fffe    ;interrupt vector location
```

```
;*******************************************************************************
;
;                                   geosRoutines
;
;This file contains equates which can be used by GEOS applications.
;
;Copyright (c) 1987 Berkeley Softworks. For the sole use of registered
;GeoProgrammer owners.
;*******************************************************************************

;Jump addresses within printer drivers

InitForPrint        == $7900    ;address of InitForPrint entry (PRINTBASE)
StartPrint          == $7903    ;address of StartPrint entry
PrintBuffer         == $7906    ;address of PrintBuffer entry
StopPrint           == $7909    ;address of StopPrint entry
GetDimensions       == $790c    ;address of GetDimensions entry
PrintASCII          == $790f    ;address of PrintASCII entry
StartASCII          == $7912    ;address of StartASCII entry
SetNLQ              == $7915    ;address of SetNLQ entry

;Jump addresses within disk drivers: these are only valid for non-1541 disk
;drive types, and for the 128 version of the 1541 driver.

Get1stDirEntry      == $9030    ;returns first dir entry
GetNxtDirEntry      == $9033    ;returns next dir entry
AllocateBlock       == $9048    ;allocates specific block
ReadLink            == $904B    ;like ReadBlock, but returns only first two
                                ;bytes of block.

;MISC

BootGEOS            == $c000
ResetHandle         == $c003
InterruptMain       == $c100

;PROCESSES

InitProcesses       == $c103
RestartProcess      == $c106
EnableProcess       == $c109
BlockProcess        == $c10c
UnblockProcess      == $c10f
FreezeProcess       == $c112
UnfreezeProcess     == $c115
```

```
;GRAPHICS

HorizontalLine      == $c118
InvertLine          == $c11b
RecoverLine         == $c11e
VerticalLine        == $c121
Rectangle           == $c124
FrameRectangle      == $c127
InvertRectangle     == $c12a
RecoverRectangle    == $c12d
DrawLine            == $c130
DrawPoint           == $c133
GraphicsString      == $c136
SetPattern          == $c139
GetScanLine         == $c13c
TestPoint           == $c13f

;BACKGROUND GENERATION

BitmapUp            == $c142

;CHARACTER MANIPULATION

PutChar             == $c145
PutString           == $c148
UseSystemFont       == $c14b

;MOUSE, MENUS, & ICONS

StartMouseMode      == $c14e
DoMenu              == $c151
RecoverMenu         == $c154
RecoverAllMenus     == $c157
DoIcons             == $c15a

;UTILITIES

DShiftLeft          == $c15d
BBMult              == $c160
BMult               == $c163
DMult               == $c166
Ddiv                == $c169
DSdiv               == $c16c
Dabs                == $c16f
Dnegate             == $c172
Ddec                == $c175
ClearRam            == $c178
FillRam             == $c17b
MoveData            == $c17e
InitRam             == $c181
PutDecimal          == $c184
GetRandom           == $c187
```

```
;MISC

MouseUp             == $c18a
MouseOff            == $c18d
DoPreviousMenu      == $c190
ReDoMenu            == $c193
GetSerialNumber     == $c196
Sleep               == $c199
ClearMouseMode      == $c19c
i_Rectangle         == $c19f
i_FrameRectangle    == $c1a2
i_RecoverRectangle  == $c1a5
i_GraphicsString    == $c1a8

;BACKGROUND GENERATION

i_BitmapUp          == $c1ab

;CHARACTER MANIPULATION

i_PutString         == $c1ae
GetRealSize         == $c1b1

;UTILITIES

i_FillRam           == $c1b4
i_MoveData          == $c1b7

;Routines added later

GetString           == $c1ba
GotoFirstMenu       == $c1bd
InitTextPrompt      == $c1c0
MainLoop            == $c1c3
DrawSprite          == $c1c6
GetCharWidth        == $c1c9
LoadCharSet         == $c1cc
PosSprite           == $c1cf
EnablSprite         == $c1d2
DisablSprite        == $c1d5
CallRoutine         == $c1d8
CalcBlksFree        == $c1db
ChkDkGEOS           == $c1de
NewDisk             == $c1e1
GetBlock            == $c1e4
PutBlock            == $c1e7
SetGEOSDisk         == $c1ea
SaveFile            == $c1ed
SetGDirEntry        == $c1f0
BldGDirEntry        == $c1f3
GetFreeDirBlk       == $c1f6
```

```
WriteFile           == $c1f9
BlkAlloc            == $c1fc
ReadFile            == $c1ff
SmallPutChar        == $c202
FollowChain         == $c205
GetFile             == $c208
FindFile            == $c20b
CRC                 == $c20e
LdFile              == $c211
EnterTurbo          == $c214
LdDeskAcc           == $c217
ReadBlock           == $c21a
LdApplic            == $c21d
WriteBlock          == $c220
VerWriteBlock       == $c223
FreeFile            == $c226
GetFHdrInfo         == $c229
EnterDeskTop        == $c22c
StartAppl           == $c22f
ExitTurbo           == $c232
PurgeTurbo          == $c235
DeleteFile          == $c238
FindFTypes          == $c23b
RstrAppl            == $c23e
ToBasic             == $c241
FastDelFile         == $c244
GetDirHead          == $c247
PutDirHead          == $c24a
NxtBlkAlloc         == $c24d
ImprintRectangle    == $c250
i_ImprintRectangle  == $c253
DoDlgBox            == $c256
RenameFile          == $c259
InitForIO           == $c25c
DoneWithIO          == $c25f
DShiftRight         == $c262
CopyString          == $c265
CopyFString         == $c268
CmpString           == $c26b
CmpFString          == $c26e
FirstInit           == $c271
OpenRecordFile      == $c274
CloseRecordFile     == $c277
NextRecord          == $c27a
PreviousRecord      == $c27d
PointRecord         == $c280
DeleteRecord        == $c283
InsertRecord        == $c286
AppendRecord        == $c289
ReadRecord          == $c28c
WriteRecord         == $c28f
```

```
SetNextFree          == $c292
UpdateRecordFile     == $c295
GetPtrCurDkNm        == $c298
PromptOn             == $c29b
PromptOff            == $c29e
OpenDisk             == $c2a1
DoInlineReturn       == $c2a4
GetNextChar          == $c2a7
BitmapClip           == $c2aa
FindBAMBit           == $c2ad
SetDevice            == $c2b0
IsMseInRegion        == $c2b3
ReadByte             == $c2b6
FreeBlock            == $c2b9
ChangeDiskDevice     == $c2bc
RstrFrmDialogue      == $c2bf
Panic                == $c2c2
BitOtherClip         == $c2c5
StashRAM             == $c2c8
FetchRAM             == $c2cb
SwapRAM              == $c2ce
VerifyRAM            == $c2d1
DoRAMOp              == $c2d4

;Jump addresses within input drivers

InitMouse            == $fe80    ;address of InitMouse entry (MOUSE_JMP)
SlowMouse            == $fe83    ;address of SlowMouse entry
UpdateMouse          == $fe86    ;address of UpdateMouse entry
SetMouse             == $fe89    ;address of SetMouse entry (128 only!)
```

```
;*************************************************************************
;
;                       geosMacros
;
;       This file contains some macro definitions which can be used by
;       GEOS applications.
;
;Copyright (c) 1987 Berkeley Softworks. For the sole use of registered
;GeoProgrammer owners.
;*************************************************************************

;*************************************************************************
;
;       Load Byte:   LoadB  dest,value
;
;       Args:       dest - address of byte to load with value
;                   value - byte to load
;
;       Action:     Load a byte with a value
;
;*************************************************************************
.macro  LoadB   dest,value
        lda     #value              ;load value
        sta     dest                ;store it
.endm

;*************************************************************************
;
;       Load Word:   LoadW  dest,value
;
;       Args:       dest - address of word to load with value
;                   value - word to load
;
;       Action:     Load a word with a value
;
;*************************************************************************
.macro  LoadW   dest,value
        lda     #](value)           ;get higher byte of value to load
        sta     dest+1              ;store it
        lda     #[(value)           ;get lower byte of value to load
        sta     dest+0              ;store it
.endm
```

```
;*************************************************************************
;
;       Move Byte:   MoveB  source,dest
;
;       Args:    source - source address
;                dest - destination address
;
;       Action:  Moves byte contents of source to destination.
;
;*************************************************************************
.macro  MoveB   source,dest
        lda     source          ;load data from source
        sta     dest            ;store it in destination
.endm

;*************************************************************************
;
;       Move Word:   MoveW  source,dest
;
;       Args:    source - source address
;                dest - destination address
;
;       Action:  Moves a word from source address to dest address.
;
;*************************************************************************
.macro  MoveW   source,dest
        lda     source+1        ;get high byte
        sta     dest+1          ;store it
        lda     source+0        ;get low byte
        sta     dest+0          ;store it
.endm

;*************************************************************************
;
;       Add Byte:   add  source
;
;       Args:    source - address of byte to add, or immediate value
;
;       Action:  a = a + source
;
;*************************************************************************
.macro  add     source
        clc
        adc     source
.endm
```

```
;*************************************************************************
;
;       Add Bytes:   AddB  source,dest
;
;       Args:    source - address of byte to add
;                dest - address of byte to add to
;
;       Action:  dest = dest + source
;
;*************************************************************************
.macro  AddB    source,dest
        clc                     ;must add with carry
        lda     source          ;get source byte
        adc     dest            ;add to destination byte
        sta     dest            ;store result
.endm
;*************************************************************************
;
;       Add Words:   AddW  source,dest
;
;       Args:    source - address of word to add
;                dest - address of word to add to
;
;       Action:  dest = dest + source
;
;*************************************************************************
.macro  AddW    source,dest
        lda     source          ;get source low byte
        clc
        adc     dest+0          ;add to destination low byte
        sta     dest+0          ;store result, sec carry with overflow
        lda     source+1        ;get source high byte
        adc     dest+1          ;add with carry to high byte dest
        sta     dest+1          ;store result
.endm
;*************************************************************************
;
;       Add Value To Byte:   AddVB  value,dest
;
;       Args:    value - constant to add to dest
;                dest - address of byte to add to
;
;       Action:  dest = dest + value
;
;*************************************************************************
.macro  AddVB   value,dest
        lda     dest
        clc
        adc     #value
        sta     dest
.endm
```

```
;*************************************************************************
;
;       Add Value to Word:   AddVW  value,dest
;
;       Args:     value - constant to add to dest
;                 dest - address of word to add to
;
;       Action:   dest = dest + value
;
;*************************************************************************
.macro  AddVW  value,dest
        clc                         ;must add with carry
        lda       #[(value)         ;get low byte of value
        adc       dest+0            ;add to low byte of word
        sta       dest+0            ;store updated value

.if     (value >= 0) && (value <= 255)
        bcc       noInc             ;carry was set if adc above overflowed.
        inc       dest+1            ;increment high byte of word
noInc:
.else
        lda       #](value)         ;carry was set if adc above overflowed.
        adc       dest+1            ;add carry + 0 to high byte of address
        sta       dest+1            ;store result
.endif

.endm

;*************************************************************************
;
;       Subtract Byte:   sub  source
;
;       Args:     source - address of byte to subtract, or immediate value
;
;       Action:   a = a - source
;
;*************************************************************************
.macro  sub       source
        sec
        sbc       source
.endm
```

```
;*************************************************************************
;
;       Sub Bytes:   SubB  source,dest
;
;       Args:    source - address of byte to subtract
;                dest - address of byte to subtract from
;
;       Action:  dest = dest - source
;
;*************************************************************************
.macro  SubB    source,dest
        sec                     ;must add with carry
        lda     dest            ;get destination byte
        sbc     source          ;subtract source byte
        sta     dest            ;store result
.endm

;*************************************************************************
;
;       Sub Words:   SubW  source,dest
;
;       Args:    source - address of byte to subtract
;                dest - address of byte to subtract from
;
;       Action:  dest = dest - source
;
;*************************************************************************
.macro  SubW    source,dest
        lda     dest+0          ;get source low byte
        sec
        sbc     source+0        ;subtract from destination low byte
        sta     dest+0          ;store result, clc carry with overflow
        lda     dest+1          ;get source high byte
        sbc     source+1        ;sub with carry from destination high byte
        sta     dest+1          ;store result
.endm

;*************************************************************************
;
;       Compare Bytes:   CmpB  source,dest
;
;       Args:    source - address of first byte
;                dest - address of second byte
;
;       Action:  compare contents of source byte to contents of dest. byte
;
;*************************************************************************
.macro  CmpB    source,dest
        lda     source          ;get source byte
        cmp     dest            ;compare source to dest
.endm
```

```
;*****************************************************************
;
;       Compare Byte To Value:  CmpBI  source,immed
;
;       Args:   source - address of first byte
;               immed - value to compare to
;
;       Action: compares contents of source to value
;
;*****************************************************************
.macro  CmpBI   source,immed
        lda     source          ;get source byte
        cmp     #immed          ;compare source to immediate value
.endm

;*****************************************************************
;
;       Compare Words:  CmpW  source,dest
;
;       Args:   source - address of first word
;               dest - address of second word
;
;       Action: compare contents of source word to contents of dest. word
;
;*****************************************************************
.macro  CmpW    source,dest
        lda     source+1        ;get high source byte
        cmp     dest+1          ;compare source to dest
        bne     done            ;need to do low byte?
        lda     source+0        ;do low byte
        cmp     dest+0          ;compare low byte
done:
.endm

;*****************************************************************
;
;       Compare Word To Value:  CmpWI  source,immed
;
;       Args:   source - address of first word
;               immed - value to compare to
;
;       Action: compares contents of source to value
;
;*****************************************************************
.macro  CmpWI   source,immed
        lda     source+1        ;get high byte
        cmp     #](immed)       ;test high byte of immediate value
        bne     done            ;don't need to do low byte
        lda     source+0        ;test low byte
        cmp     #[(immed)
done:
.endm
```

```
;*************************************************************************
;
;       Reset Bit:   rmb  bitNumber,dest
;
;       Args:    bitNumber - bit number in byte to reset (7 for MSD, 0 for LSD)
;                dest - address of byte which contains bit to reset
;
;       Action:  resets bit in destination byte
;                fast version (rmbf) trashes the accumulator
;
;*************************************************************************
.macro  rmb      bitNumber,dest
        pha
        lda      #[~(1 << bitNumber)
        and      dest
        sta      dest
        pla
.endm

.macro  rmbf     bitNumber,dest
        lda      #[~(1 << bitNumber)
        and      dest
        sta      dest
.endm

;*************************************************************************
;
;       Branch on Bit Set:   bbs  bitNumber,source,addr
;
;       Args:    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
;
;       Action:  tests bit in source byte, branches if is set
;                fast version (bbsf) trashes the accumulator
;
;*************************************************************************
.macro  bbs      bitNumber,source,addr
        php
        pha
        lda      source
        and      #(1 << bitNumber)
        beq      nobranch
        pla
        plp
        bra      addr
nobranch:
        pla
        plp
.endm
```

```
.macro  bbsf    bitNumber,source,addr
.if     (bitNumber = 7)
        bit     source
        bmi     addr
.elif   (bitNumber = 6)
        bit     source
        bvs     addr
.else
        lda     source
        and     #(1 << bitNumber)
        bne     addr
.endif
.endm

;**************************************************************************
;
;       Branch on Bit Reset:    bbr  bitNumber,source,addr
;
;       Args:   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
;
;       Action: tests bit in source byte, branches if is reset
;               fast version (bbsf) trashes the accumulator
;
;**************************************************************************
.macro  bbr     bitNumber,source,addr
        php
        pha
        lda     source
        and     #(1 << bitNumber)
        bne     nobranch
        pla
        plp
        bra     addr
nobranch:
        pla
        plp
.endm

.macro  bbrf    bitNumber,source,addr
.if     (bitNumber = 7)
        bit     source
        bpl     addr
.elif   (bitNumber = 6)
        bit     source
        bvc     addr
.else
        lda     source
        and     #(1 << bitNumber)
        beq     addr
.endif
.endm
```

```
;*************************************************************************
;
;                       SamSeq
;
;       This is the main file for the GeoProgrammer package sample
;       application. It contains all of the code and data required
;       for assembly.
;
;Copyright (c) 1987 Berkeley Softworks. For the sole use of registered
;GeoProgrammer owners.
;*************************************************************************

.if     Pass1                   ;Only need to include these files
                                ;during assembler's first pass.
.include geosSym                ;get GEOS definitions
.include geosMac                ;get GEOS macro definitions
.endif

;Our program starts here. The first thing we do is clear the screen and
;initialize our menus and icons. Then we RTS to GEOS mainloop.
;When an event happens, such as the user selects a menu item or one of our
;icons, GEOS will call one of our handler routines.

        .psect                  ;program code section starts here
                                ;(GeoLinker will give this an address of $0400)

ProgStart:

        LoadB   dispBufferOn,# (ST_WR_FORE | ST_WR_BACK)
                                ;allow writes to foreground and background

        LoadW   r0,#ClearScreen ;point to graphics string to clear screen
        jsr     GraphicsString

        LoadW   r0,#MenuTable   ;point to menu definition table
        lda     #0              ;place cursor on first menu item when done
        jsr     DoMenu          ;have GEOS draw the menus on the screen

        LoadW   r0,#IconTable   ;point to icon definition table
        jsr     DoIcons         ;have GEOS draw the icons on the screen
        rts
```

```
;Here are some data tables for the init code shown above:

ClearScreen:                            ;graphics string table to clear screen
        .byte   NEWPATTERN,2            ;set new pattern value
        .byte   MOVEPENTO               ;move pen to:
        .word   0                       ;top left corner of screen
        .byte   0
        .byte   RECTANGLETO             ;draw filled rectangle to bottom right corner
        .word   319
        .byte   199
        .byte   NULL                    ;end of application

MenuTable:                              ;menu definition table for main horizontal menu
        .byte   0,14                    ;top and bottom y coordinates
        .word   0,49                    ;left and right x coordinates
        .byte   2 | HORIZONTAL          ;number of menu items, type of menu

        .word   GeosText                ;pointer to text for menu item
        .byte   VERTICAL                ;type of menu
        .word   GeosSubMenu             ;pointer to menu structure

        .word   FileText                ;pointer to text for menu item
        .byte   VERTICAL                ;type of menu
        .word   FileSubMenu             ;pointer to menu structure

GeosSubMenu:                            ;menu definition table for GEOS vertical menu
        .byte   15,30                   ;top and bottom y coordinates
        .word   0,79                    ;left and right x coordinates
        .byte   1 | VERTICAL            ;number of menu items, type of menu

        .word   AboutText               ;pointer to text for menu item
        .byte   MENU_ACTION             ;type of action
        .word   DoAbout                 ;pointer to handler routine

FileSubMenu:                            ;menu definition table for FILE vertical menu
        .byte   15,44                   ;top and bottom y coordinates
        .word   29,64                   ;left and right x coordinates
        .byte   2 | VERTICAL            ;number of menu items, type of menu

        .word   CloseText               ;pointer to text for menu item
        .byte   MENU_ACTION             ;type of action
        .word   DoClose                 ;pointer to handler routine

        .word   QuitText                ;pointer to text for menu item
        .byte   MENU_ACTION             ;type of action
        .word   DoQuit                  ;pointer to handler routine
```

```
;text strings for above menus

GeosText:
        .byte   "geos",0
FileText:
        .byte   "file",0
AboutText:
        .byte   "SampleSeq info",0
CloseText:
        .byte   "close",0
QuitText:
        .byte   "quit",0

;icon definition table

IconTable:
        .byte   1               ;number of icons
        .word   0               ;x position to place mouse at when done
        .byte   0               ;y position to place mouse at when done

        .word   Icon1Picture    ;pointer to compacted bitmap for icon
        .byte   3               ;x position in bytes
        .byte   60              ;y position in scanlines
        .byte   ICON_1_WIDTH    ;width of icon in bytes
        .byte   ICON_1_HEIGHT   ;height of icon in scanlines
        .word   DoIcon1         ;pointer to handler routine

Icon1Picture:                   ;assembler will place compacted bitmap data
                                ;here for this picture:

                                    +------+
                                    | Icon |
                                    +------+

ICON_1_WIDTH  = picW            ;store bitmap size values for use in above
ICON_1_HEIGHT = picH            ;table on pass 2. (picW and picH are set by
                                ;the assembler.)
```

```
;Event handler routines: are called by GEOS when an event happens,
;such as user selecting a menu item or clicking on an icon.

DoAbout:
        jsr     GotoFirstMenu       ;roll menu back up

        ;code to handle this event goes here

        rts                         ;all done

DoClose:
        jsr     GotoFirstMenu       ;roll menu back up

        ;code to handle this event goes here

        rts                         ;all done

DoQuit:
        jsr     GotoFirstMenu       ;roll menu back up
        jmp     EnterDeskTop        ;return to deskTop!

DoIcon1:
        ;code to handle this event goes here
        rts
```

```
;*************************************************************************
;
;                       SamSeqHdr
;
;       This file contains the header block definition for the GeoProgrammer
;       package sample sequential application.
;
;Copyright (c) 1987 Berkeley Softworks. For the sole use of registered
;GeoProgrammer owners.
;*************************************************************************

.if     Pass1                   ;Only need to include this file
                                ;during assembler's first pass.
.include geosSym                ;get GEOS definitions
.endif

;Here is our header. The SamSeq.lnk file will instruct the linker
;to attach it to our sample application.

        .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:

                                        Seq

        .byte   $80 | USR       ;Commodore file type, with bit 7 set.
        .byte   APPLICATION     ;Geos file type
        .byte   SEQUENTIAL      ;Geos file structure type
        .word   ProgStart       ;start address of program (where to load to)
        .word   $3ff            ;usually end address, but only needed for
                                ;desk accessories.
        .word   ProgStart       ;init address of program (where to JMP to)
        .byte   "SampleSeq  V1.0",0,0,0,$00
                                ;permanent filename: 12 characters,
                                ;followed by 4 character version number,
                                ;followed by 3 zeroes,
                                ;followed by 40/80 column flag.
        .byte   "Eric E. Del Sesto  ",0
                                ;twenty character author name

        ;end of header section which is checked for accuracy
        .block  160-117         ;skip 43 bytes...
        .byte   "This is the GeoProgrammer sample "
        .byte   "sequential GEOS application.",0
        .endh
```

```
;*************************************************************************
;
;                           SamSeq.lnk
;
;       This is the GeoLinker command file for the GeoProgrammer package
;       sample application.
;
;Copyright (c) 1987 Berkeley Softworks. For the sole use of registered
;GeoProgrammer owners.
;*************************************************************************

.output   SampleSeq       ;name for output file
.header   SamSeqHdr.rel   ;name of file containing header block to use

.seq                      ;this is a sequential application

.psect    $0400           ;program code starts at $0400
.ramsect  $5000           ;program data area starts at $5000

SamSeq.rel                ;name of file which contains relocatable
                          ;code and data from GeoAssembler
```

```
;*************************************************************************
;
;                       SampleSeq.dbm
;
;       This is file contains GeoDebugger macro definitions for use when
;       debugging the SampleSeq application.
;
;Copyright (c) 1987 Berkeley Softworks. For the sole use of registered
;GeoProgrammer owners.
;*************************************************************************

;This "autoexec" macro will run when the GeoDebugger starts up.
;It sets one of the debugger option flags.

.macro autoexec                 ;name of macro
poff[cr]                        ;turn printing off
opt 4,0[cr]                     ;disable option 4 (case distinction)
.endm                           ;end of macro

;The following macro causes the debugger to step once and display the results.

.macro sr                       ;name of macro
s[cr]                           ;step one instruction
pr[cr]                          ;print blank line
r[cr]                           ;print registers
pr "----------------------------------------"[cr]
.endm                           ;end of macro

;This macro changes the characters in the "GEOS" menu to upper-case.

.macro geos
m GeosText[cr]                  ;open location as memory
[sp]"GEOS"[cr]                  ;deposit new string
.endm
```

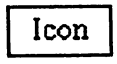
```
;*************************************************************************
;
;                       SamVlirRes
;
;       This is the main file for the GeoProgrammer package sample
;       VLIR application. It contains all of the code and data required
;       for assembling the resident portion of the program.
;       See the GeoProgrammer User's Guide for a roadmap to this program.
;
;Copyright (c) 1987 Berkeley Softworks. For the sole use of registered
;GeoProgrammer owners.
;*************************************************************************

.if     Pass1                   ;Only need to include these files
                                ;during assembler's first pass.
.include geosSym                ;get GEOS definitions
.include geosMac                ;get GEOS macro definitions
.endif

;Now include our program's equates and zero page variables:
;(We could let the linker handle the equates, but we MUST include the
;zero page variables here so that addressing modes can be resolved.)

.if     Pass1                   ;Only need to include these files
                                ;during assembler's first pass.
.include SamVlirEquates         ;get sample VLIR equates
.include SamVlirZPVars          ;get sample VLIR zero page variables
.endif

;The resident portion of our program starts here. The first thing we do is
;clear the screen and initialize our menus and icons. Then we RTS to GEOS
;mainloop. When an event happens, such as the user selects a menu item or
;one of our icons, GEOS will call one of our handler routines.

        .psect                  ;program code section starts here
                                ;(GeoLinker will give this an address of $0400)

ResStart:
        LoadB   dispBufferOn,# (ST_WR_FORE | ST_WR_BACK)
                                ;allow writes to foreground and background

        LoadW   r0,#ClearScreen ;point to graphics string to clear screen
        jsr     GraphicsString

        jsr     InitSwap        ;set up table to facilitate module swapping
                                ;later on

        jsr     InitDA          ;set up GEOS menu to facilitate running of
                                ;desk accessories later on
```

```
        LoadW   r0,#MenuTable       ;point to menu definition table
        lda     #0                  ;position mouse on first menu item
        jsr     DoMenu              ;have GEOS draw the menus on the screen

        LoadW   r0,#IconTable       ;point to icon definition table
        jsr     DoIcons             ;have GEOS draw the icons on the screen
        rts

;Here are some data tables for the init code shown above:

ClearScreen:                        ;graphics string table to clear screen
        .byte   NEWPATTERN,2        ;set new pattern value
        .byte   MOVEPENTO           ;move pen to:
        .word   0                   ;top left corner of screen
        .byte   0
        .byte   RECTANGLETO         ;draw filled rectangle to bottom right corner
        .word   319
        .byte   199
        .byte   NULL                ;end of application

MenuTable:
        .byte   MM_TOP              ;top of menu
        .byte   MM_BOTTOM           ;bottom of menu
        .word   MM_LEFT             ;left side
        .word   MM_RIGHT            ;right side
        .byte   MM_COUNT | HORIZONTAL
                                    ;number of menu items, type of menu

        .word   GeosText            ;pointer to text for menu item
        .byte   VERTICAL            ;type of menu
        .word   GeosSubMenu         ;pointer to menu structure

        .word   FileText            ;pointer to text for menu item
        .byte   VERTICAL            ;type of menu
        .word   FileSubMenu         ;pointer to menu structure

        .word   EditText            ;pointer to text for menu item
        .byte   VERTICAL            ;type of menu
        .word   EditSubMenu         ;pointer to menu structure
```

```
;Note: the GEOS sub-menu as it appears below is constructed assuming
;only 1 item- the "SampleVlir info" item. When the application is started,
;a routine called "InitDA" will update this structure according to the
;number of desk accessories found on the application disk.

GeosSubMenu:                    ;menu definition table for GEOS vertical menu
        .byte   SM_TOP          ;top scanline #
        .byte   SM_TOP+1+(1*14) ;bottom scanline #
        .word   GM_LEFT         ;left x position
        .word   GM_LEFT + GM_WIDTH
                                ;right x position
        .byte   VERTICAL | GM_COUNT
                                ;number of menu items, type of menu

        .word   AboutText       ;pointer to text for menu item
        .byte   MENU_ACTION     ;type of action
        .word   R_DoAbout       ;pointer to handler routine
                                ;(R_ means routine is resident)

        .word   DA0Text         ;pointer to text for menu item
        .byte   MENU_ACTION     ;type of action
        .word   R_RunDA         ;pointer to handler routine

        .word   DA1Text
        .byte   MENU_ACTION
        .word   R_RunDA

        .word   DA2Text
        .byte   MENU_ACTION
        .word   R_RunDA

        .word   DA3Text
        .byte   MENU_ACTION
        .word   R_RunDA

        .word   DA4Text
        .byte   MENU_ACTION
        .word   R_RunDA

        .word   DA5Text
        .byte   MENU_ACTION
        .word   R_RunDA

        .word   DA6Text
        .byte   MENU_ACTION
        .word   R_RunDA

        .word   DA7Text
        .byte   MENU_ACTION
        .word   R_RunDA
```

```
FileSubMenu:                        ;menu definition table for FILE vertical menu
        .byte   SM_TOP              ;top scanline #
        .byte   SM_TOP+1+(FM_COUNT*14)
                                    ;bottom scanline #
        .word   FM_LEFT             ;left x position
        .word   FM_LEFT + FM_WIDTH
                                    ;right x position
        .byte   VERTICAL | FM_COUNT
                                    ;number of menu items, type of menu

        .word   CloseText           ;pointer to text for menu item
        .byte   MENU_ACTION         ;type of action
        .word   R_DoClose           ;pointer to handler routine
                                    ;(R_ means routine is resident)

        .word   QuitText            ;pointer to text for menu item
        .byte   MENU_ACTION         ;type of action
        .word   R_DoQuit            ;pointer to handler routine

EditSubMenu:                        ;menu definition table for FILE vertical menu
        .byte   SM_TOP              ;top scanline #
        .byte   SM_TOP+1+(EM_COUNT*14)
                                    ;bottom scanline #
        .word   EM_LEFT             ;left x position
        .word   EM_LEFT + EM_WIDTH
                                    ;right x position
        .byte   VERTICAL | EM_COUNT
                                    ;number of menu items, type of menu

        .word   CutText             ;pointer to text for menu item
        .byte   MENU_ACTION         ;type of action
        .word   R_DoCut             ;pointer to handler routine
                                    ;(R_ means routine is resident)

        .word   CopyText
        .byte   MENU_ACTION
        .word   R_DoCopy

        .word   PasteText
        .byte   MENU_ACTION
        .word   R_DoPaste

;Text strings for above menu definitions

GeosText:
        .byte   "geos",0
FileText:
        .byte   "file",0
EditText:
        .byte   "edit",0
AboutText:
        .byte   "SampleVlir info",0
```

```
;The following text strings are updated by the InitDA routine.
;They will contain filenames of all the desk accessories found on the
;application disk.

DA0Text: .byte    "desk accessory 0",0
DA1Text: .byte    "desk accessory 1",0
DA2Text: .byte    "desk accessory 2",0
DA3Text: .byte    "desk accessory 3",0
DA4Text: .byte    "desk accessory 4",0
DA5Text: .byte    "desk accessory 5",0
DA6Text: .byte    "desk accessory 6",0
DA7Text: .byte    "desk accessory 7",0

CloseText:
        .byte     "close",0
QuitText:
        .byte     "quit",0
CutText:
        .byte     "cut",0
CopyText:
        .byte     "copy",0
PasteText:
        .byte     "paste",0

IconTable:                          ;icon definition table
        .byte     1                 ;number of icons
        .word     0                 ;x position to place mouse at when done
        .byte     0                 ;y position to place mouse at when done

        .word     Icon1Picture      ;pointer to compacted bitmap for icon
        .byte     3                 ;x position in bytes
        .byte     60                ;y position in scanlines
        .byte     ICON_1_WIDTH      ;width of icon in bytes
        .byte     ICON_1_HEIGHT     ;height of icon in scanlines
        .word     R_DoIcon1         ;pointer to handler routine
                                    ;(R_ means routine is resident)

Icon1Picture:                       ;assembler will place compacted bitmap data
                                    ;here for this picture:
```




```
ICON_1_WIDTH  = picW                ;store bitmap size values for use in above
ICON_1_HEIGHT = picH                ;table on pass 2. (picW and picH are set by
                                    ;the assembler.)
```

```
;*************************************************************************
;               R_DoAbout, R_RunDA, R_DoClose, R_DoQuit,
;               R_DoCut, R_DoCopy, R_DoPaste, R_DoIcon1
;
;       These routines are all Resident Handler Routines. They are called
;       by GEOS when an event happens, such as the user selecting a menu
;       item or clicking on an icon. All of these routines (except C_DoQuit)
;       load in a swap module before calling their handler routine in that
;       module. Since R_DoQuit is a small routine, it does not have to swap
;       in a module; all the necessary code to quit the application is
;       resident.
;
;       Author:  Eric E. Del Sesto, August 1987
;       Caller:  GEOS menu or icon dispatch handlers
;       Pass:    if from menu: a = sub-menu item number
;       Returns: nothing
;       Alters:  a, x, y, r0-r15 (probably)
;
;*************************************************************************
R_DoAbout:
        jsr     GotoFirstMenu       ;roll menu back up

        ;code to handle this event goes here

        rts                         ;all done

R_RunDA:
        pha                         ;save sub-menu number
        jsr     GotoFirstMenu       ;roll menu back up
        jsr     FileIn              ;swap File module into swap area
        pla                         ;recall sub-menu number
        jsr     J_RunDA             ;call DA handling routine in File module
                                    ;(J_RunDA is in jump table in the
                                    ;SamVlirEquates file).
        rts                         ;return to GEOS mainloop

R_DoClose:
        jsr     GotoFirstMenu       ;roll menu back up
        jsr     FileIn              ;swap File module into swap area
        jsr     J_DoClose           ;call handling routine in File module
        rts                         ;return to GEOS mainloop

R_DoQuit:
        jsr     GotoFirstMenu       ;roll menu back up
        jmp     EnterDeskTop        ;return to deskTop!

R_DoCut:
        jsr     GotoFirstMenu       ;roll menu back up
        jsr     EditIn              ;swap Edit module into swap area
        jsr     J_DoCut             ;call handling routine in Edit module
        rts                         ;return to GEOS mainloop
```

```
R_DoCopy:
        jsr     GotoFirstMenu
        jsr     EditIn
        jsr     J_DoCopy
        rts

R_DoPaste:
        jsr     GotoFirstMenu
        jsr     EditIn
        jsr     J_DoPaste
        rts

R_DoIcon1:
        jsr     EditIn          ;swap Edit module into swap area
        jsr     J_DoIcon1       ;call handling routine in Edit module
        rts                     ;return to GEOS mainloop

;This routine swaps the file module in.

FileIn:
        lda     #MOD_FILE       ;get number of file module
        jsr     SwapMod         ;call swap routine to bring module in
        rts

;This routine swaps the edit module in.

EditIn:
        lda     #MOD_EDIT       ;get number of edit module
        jsr     SwapMod         ;call swap routine to bring module in
        rts
```

```
;*************************************************************************
;                       InitSwap
;
;       This routine sets up a table which contains the track and sector
;       numbers for each of the program modules which can be loaded.
;       This table will be used by the SwapMod routine later on.
;
;       Author:  Tony / Eric, August 1987
;       Caller:  ResStart
;       Pass:    application disk opened
;       Returns: appName = filename of application file
;                swapTable = table of (T,S) pairs, one for
;                each module. See NUM_MODS equate.
;                curModule = $ff (no module currently loaded)
;       Alters:  a, x, y, r0-r2, r4-r7, r10
;
;*************************************************************************
InitSwap:
        ;This first step is in case someone has changed the application's
        ;filename: we search the disk using the application permanent name,
        ;and find out what the filename is.

        LoadW   r6,#appName         ;point to buffer to store filename in
        LoadB   r7L,#APPLICATION ;look for files of type application
        LoadB   r7H,#1              ;only want 1 file, the application
        LoadW   r10,#NameString     ;point to application's permanent name
        jsr     FindFTypes          ;GEOS system call to do directory search
                                    ;for above. Assume no errors...

        ;appName has filename now. Open the application file as a VLIR file.

        LoadW   r0,#appName         ;set up ptr to filename as is on disk
        jsr     OpenRecordFile      ;initialize for reading records as VLIR.

        ;fileHeader now contains index table for application file.
        ;Copy track/sector pointers into table that will be used by SwapMod
        ;routine to load modules. (i_MoveData not used to simplify stepping.)

        LoadW   r0,#fileHeader+4    ;source in fileHeader
        LoadW   r1,#swapTable       ;destination, to hold index table data
        LoadW   r2,#NUM_MODS*2 ;number of bytes to copy
        jsr     MoveData            ;use GEOS MoveData routine

        jsr     CloseRecordFile     ;and close application file
                                    ;(assume no errors)
        ;curModule is a variable which contains the number of the currently loaded
        ;module. Initialize it to a value which won't match any number.
        LoadB   curModule,#$ff
        rts                         ;all done

NameString:     ;permanent name string for our application
        .byte   "SampleVlir  V1.0",0
```

```
;*************************************************************************
;                       InitDA
;
;       This routine builds out the GEOS menu item table so that it contains
;       the names of the desk accessories on the disk. Also see the RunDA
;       routine.
;
;       Author:  Tony / Eric, August 1987
;       Caller:  ResStart
;       Pass:    application disk opened
;       Returns: GEOS menu structure updated
;       Alters:  a, x, y, r0-r2, r4, r6, r7, r10
;
;*************************************************************************
InitDA:
        ;first have GEOS search disk for files which have a GEOS type
        ;of DESK_ACC. Copy their names into the menu structure.

        LoadW   r6,#DA0Text     ;put filenames in array for menu text
        LoadB   r7L,#DESK_ACC   ;look for files of type desk accessory
        LoadB   r7H,#NUM_DA     ;maximum of 7 desk accessories may be listed
        LoadW   r10,#0          ;don't care about permanent names
        jsr     FindFTypes      ;call GEOS routine

        ;now calculate the number of desk accessories found and update
        ;some more crucial bytes in the menu structure.

        lda     #NUM_DA         ;r7H returned with (7 - num files found)
        sub     r7H             ;subtract from 7 to get number of files
        beq     90$             ;exit if there are no files...

        clc
        adc     #1              ;add one for "SampleVlir info" menu item
        pha                     ;save the number of menu items
        ora     #VERTICAL       ;and "or" with VERTICAL flag
        sta     GeosSubMenu+6   ;to set new number of sub-menu items.
        pla                     ;recover number of menu items

        ;now calculate height of menu in scanlines: is 14 per menu item.

        sta     r0L             ;save in temp register
        asl     a               ;multiply by 16
        asl     a
        asl     a
        asl     a
        sub     r0L             ;and subtract itself twice to get
        sub     r0L             ;final result of numItems * 14
        clc
        adc     #SM_TOP+1       ;add to top scanline number of sub-menu
        sta     GeosSubMenu+1   ;set new bottom for sub-menu

90$:    rts                     ;all done
```

```
;*************************************************************************
;                       SwapMod
;
;       This routine swaps a module in. Note how it uses "ReadFile" instead
;       of "ReadRecord" so that it does not affect any opened VLIR file.
;
;       Author:  Eric E. Del Sesto, August 1987
;       Caller:  top-level resident routines
;       Pass:    a = number of module to swap in
;                curModule = number of module which is currently in
;       Returns: curModule = number of module which is swapped in
;       Alters:  a, x, y, r1-r13
;
;*************************************************************************
SwapMod:
        cmp     curModule       ;see if module is already swapped in
        beq     90$             ;skip to end if so...
        sta     curModule       ;save new module number

        ;now use module number to get track and sector information on
        ;record which contains module.

        sec
        sbc     #1              ;subtract 1 and multiply by 2
        asl     a               ;to get index to (T,S) word.
        tay                     ;because of word length entries in swapTable

        lda     swapTable+0,y   ;get track number
        sta     r1L
        lda     swapTable+1,y   ;get sector number
        sta     r1H

        ;load module into swap area

        LoadW   r7,#SWAP_BASE   ;base address for load
        LoadW   r2,#SWAP_SIZE   ;maximum size of module
        jsr     ReadFile        ;read the record in
                                ;(You may want to check for errors here.)

90$:    ;all done
        rts
```

```
;*************************************************************************
;                       SwapMod
;
;       This routine swaps a module in. Note how it uses "ReadFile" instead
;       of "ReadRecord" so that it does not affect any opened VLIR file.
;
;       Author:  Eric E. Del Sesto, August 1987
;       Caller:  top-level resident routines
;       Pass:    a = number of module to swap in
;                curModule = number of module which is currently in
;       Returns: curModule = number of module which is swapped in
;       Alters:  a, x, y, r1-r13
;
;*************************************************************************
SwapMod:
        cmp     curModule       ;see if module is already swapped in
        beq     90$             ;skip to end if so...
        sta     curModule       ;save new module number

        ;now use module number to get track and sector information on
        ;record which contains module.

        sec
        sbc     #1              ;subtract 1 and multiply by 2
        asl     a               ;to get index to (T,S) word.
        tay                     ;because of word length entries in swapTable

        lda     swapTable+0,y   ;get track number
        sta     r1L
        lda     swapTable+1,y   ;get sector number
        sta     r1H

        ;load module into swap area

        LoadW   r7,#SWAP_BASE   ;base address for load
        LoadW   r2,#SWAP_SIZE   ;maximum size of module
        jsr     ReadFile        ;read the record in
                                ;(You may want to check for errors here.)

90$:    ;all done
        rts
```

```
;*************************************************************************
;                       Global Variables
;
;           These variables are resident and thus always accessable by any module
;           of our application.
;
;*************************************************************************

            .ramsect
                                        ;variable section starts here
                                        ;(GeoLinker will give this an address of $5000)

swapTable:
            .block      NUM_MODS*2      ;holds (T,S) pairs, one for each module
                                        ;of our application

appName:
            .block      17              ;holds application filename. Really only
                                        ;necessary during initialization.
```

```
;*************************************************************************
;
;                       SamVlirFile
;
;       This file contains the File Module code for the GeoProgrammer
;       package sample VLIR application. It contains all of the code
;       and data required for assembling the File Module portion of the program.
;
;Copyright (c) 1987 Berkeley Softworks. For the sole use of registered
;GeoProgrammer owners.
;*************************************************************************

;Now include GEOS definitions and our definitions:
;(We could let the linker handle this, but doing it here speeds up the
;link process. We MUST include the zero page variables here so that
;addressing modes can be resolved.)

.if     Pass1                   ;Only need to include these files
.noeqin                         ;during assembler's first pass.
.noglbl
.include geosSym                ;get GEOS definitions
.include geosMac                ;get GEOS macro definitions
.include SamVlirEquates         ;get sample VLIR equates
.include SamVlirZPVars          ;get sample VLIR zero page variables
.eqin
.glbl
.endif

;The File module starts here with a jump table so the resident portion
;of our code can JSR to routines in this module without knowing their
;exact address. See the jump table equates in the SamVlirEquates file.

        .psect                  ;module code section starts here
                                ;(GeoLinker will give this an address
                                ;SWAP_BASE, which is $1000.)

FileMod:
        jmp     RunDA           ;first jump table entry
        jmp     DoClose         ;2nd
```

```
;*************************************************************************
;                       RunDeskAccessory
;
;       This routine loads and runs a desk accessory. Note that the call
;       to GetFile to load the desk accessory causes the memory under the
;       desk accessory to be swapped out and control transferred to the desk
;       accessory. When the desk accessory is "turned off" by executing a
;       call to RstrAppl, control returns to the application (in this case
;       the deskTop) immediately following the call to GetFile.
;
;       Author:  Tony Requist / Eric E. Del Sesto, August 1987
;       Caller:  GEOS mainloop when DA name in GEOS menu is selected.
;       Pass:    a = sub-menu item number
;       Returns: nothing
;       Alters:  a, x, y, r0-r15
;
;*************************************************************************
RunDA:
        ;first use the sub-menu item number to point to the filename
        ;for the desk accessory

        sta     r6L             ;Store a copy of the selected menu item's #
        asl     a               ;The menu item number times 17, added to the
        asl     a               ;base of the strings for geos submenu items
        asl     a               ;(each 17 bytes apart) gives the address of
        asl     a               ;the filename for this DA.

        add     r6L             ;now have menu item number times 17
        clc
        adc     #[(DA0Text-17)  ;add low byte of base address of table
        sta     r6L             ;save low byte into r6

        lda     #0              ;high byte of offset is 0
        adc     #](DA0Text-17)  ;add to high byte of base address of table
                                ;(considering carry from low byte)
        sta     r6H             ;save high byte

        PushW   r6              ;save pointer to desk accessory filename

        ;place code that will run before a desk accessory here

        ;close any open VLIR files

        ;copy sprite picture data (for 7 sprites) to a buffer

        LoadW   r0,#spr1pic     ;from sprite picture data area: $8a40
        LoadW   r1,#spriteBuf   ;to a (7*64) byte buffer.
        LoadW   r2,#(7*64)
        jsr     MoveData        ;move data
```

```
        ;for applications which read other drives, should use OpenDisk
        ;to open application disk here.

        PopW    r6              ;recall pointer to desk accessory filename

        ;save sprite's double-Y flag in case is changed by desk accessory

        ldx     CPU_DATA        ;save memory map status for now
        LoadB   CPU_DATA,#IO_IN;swap I/O space in
        PushB   moby2           ;save VIC's sprite double-y byte
        LoadB   moby2,#0        ;and set for "no doubling"
        stx     CPU_DATA        ;restore previous memory map

        LoadB   r0L,#0          ;use standard loading option (always 0 for DAs)
                                ;pass flag to GetFile routine

        lda     #%00000000      ;B7 = 1 to make DA save foreground screen to
                                ;buffer or disk and recover when done.
                                ;B6 = 1 to make DA save color information
                                ;to buffer or disk and recover when done.
        sta     r10L            ;pass flag to GetFile routine

        jsr     GetFile         ;load and run desk accessory.

;at this point, GEOS saves:
;       pointers to menu and icon structures
;       all sprite x, y, color, and doubleX info
;
;desk accessory code must:
;       set its own sprite pictures, (x,y) positions, colors,
;               and doubleXY information.
;       set the desired screen colors (40-column mode only)
;       not use $0200-$03ff for variables, because some
;               new applications (geoFile, geoDebug) do
;
;when desk accessory has finished, GEOS restores:
;       pointers to menu and icon structures
;       all sprite x, y, color, and doubleX info

        stx     r6L             ;save error status for now

        ;restore sprite's double-y flag in case was changed by desk accessory

        ldx     CPU_DATA        ;save memory map status for now
        LoadB   CPU_DATA,#IO_IN;swap I/O space in
        PopB    moby2           ;restore VIC's sprite double-y byte
        stx     CPU_DATA        ;restore previous memory map
```

```
                ;restore sprite picture data

                LoadW   r0,#spriteBuf       ;source
                LoadW   r1,#spr1pic         ;destination
                LoadW   r2,#(7*64)
                jsr     MoveData

                ;since we did not have DA restore our colors,
                ;must now fill color table with default screen color

                MoveB   screencolors,r2L
                LoadW   r1,#COLOR_MATRIX
                LoadW   r0,#(25*40)
                jsr     FillRam

                ;since we did not have DA save our foreground screen,
                ;must recover from background here.

                LoadB   r2L,#MM_BOTTOM+1
                                            ;top y coordinate (do not restore menu area-
                                            ;DAs cannot affect it.)
                LoadB   r2H,#199            ;bottom y coordinates
                LoadW   r3,#0               ;left x coordinate
                LoadW   r4,#319             ;right x coordinates
                jsr     RecoverRectangle

;On error handling: any error that happened must be related to loading
;the desk accessory. Might want to distinguish between INSUFF_SPACE
;and other disk errors.

                ldx     r6L                 ;get error number
                beq     20$                 ;skip if no error...

                ;handle errors here

20$:            ;code to run after desk accessory completion goes here

                ;re-open VLIR files here

                rts                         ;return to resident R_RunDA routine, which
                                            ;will return to GEOS mainloop, letting
                                            ;application continue...
```

```
;*************************************************************************
;                       DoClose
;
;       This is a dummy event routine. You can file in your own code here.
;
;       Author:  Eric E. Del Sesto, August 1987
;       Caller:  GEOS mainloop when the "close" menu item is selected
;       Pass:    nothing
;       Returns: nothing
;       Alters:  a, x, y, r0-r15
;
;*************************************************************************
DoClose:
        jsr     GotoFirstMenu
        rts

;*************************************************************************
;                       File Module local variables
;
;       This area contains definitions for variables which are local to
;       the File Module. No other module (including resident) can access
;       these variables. These variables are trashed whenever the File
;       Module swaps in or out, and so cannot be used for anything more
;       than temporary storage for routines in this module.
;
;*************************************************************************

        .ramsect
                                ;variable section starts here
                                ;(GeoLinker will give this an address
                                ;of SWAP_VARS, which is $1f00.)

spriteBuf:
        .block  7 * 64          ;holds 7 sprite images (#1-#8) while
                                ;desk accessory is running.
```

```
;*************************************************************************
;
;                       SamVlirEdit
;
;       This file contains the File Module code for the GeoProgrammer
;       package sample VLIR application. It contains all of the code
;       and data required for assembling the File Module portion of the program.
;
;Copyright (c) 1987 Berkeley Softworks. For the sole use of registered
;GeoProgrammer owners.
;*************************************************************************

;Now include GEOS definitions and our definitions:
;(We could let the linker handle this, but doing it here speeds up the
;link process. We MUST include the zero page variables here so that
;addressing modes can be resolved.)

.if     Pass1                   ;Only need to include these files
.noeqin                         ;during assembler's first pass.
.noglbl
.include  geosSym               ;get GEOS definitions
.include  geosMac               ;get GEOS macro definitions
.include  SamVlirEquates        ;get sample VLIR equates
.include  SamVlirZPVars         ;get sample VLIR zero page variables
.eqin
.glbl
.endif

;The Edit Module starts here with a jump table so the resident portion
;of our code can JSR to routines in this module without knowing their
;exact address. See the jump table equates in the SamVlirEquates file.

        .psect                  ;module code section starts here
                                ;(GeoLinker will give this an address
                                ;SWAP_BASE, which is $1000.)

EditMod:
        jmp     DoCut           ;first jump table entry
        jmp     DoCopy          ;2nd
        jmp     DoPaste         ;3rd
        jmp     DoIcon1         ;4th
```

```
;*************************************************************************
;
;                       SamVlirEdit
;
;       This file contains the File Module code for the GeoProgrammer
;       package sample VLIR application. It contains all of the code
;       and data required for assembling the File Module portion of the program.
;
;Copyright (c) 1987 Berkeley Softworks. For the sole use of registered
;GeoProgrammer owners.
;*************************************************************************

;Now include GEOS definitions and our definitions:
;(We could let the linker handle this, but doing it here speeds up the
;link process. We MUST include the zero page variables here so that
;addressing modes can be resolved.)

.if     Pass1                   ;Only need to include these files
.noeqin                         ;during assembler's first pass.
.noglbl
.include geosSym                ;get GEOS definitions
.include geosMac                ;get GEOS macro definitions
.include SamVlirEquates         ;get sample VLIR equates
.include SamVlirZPVars          ;get sample VLIR zero page variables
.eqin
.glbl
.endif

;The Edit Module starts here with a jump table so the resident portion
;of our code can JSR to routines in this module without knowing their
;exact address. See the jump table equates in the SamVlirEquates file.

        .psect                  ;module code section starts here
                                ;(GeoLinker will give this an address
                                ;SWAP_BASE, which is $1000.)

EditMod:
        jmp     DoCut           ;first jump table entry
        jmp     DoCopy          ;2nd
        jmp     DoPaste         ;3rd
        jmp     DoIcon1         ;4th
```

```
;*************************************************************************
;                       DoCut
;
;       This is a dummy event handler routine. Customize this for your
;       own application.
;
;       Author:  Eric E. Del Sesto, August 1987
;       Caller:  R_DoCut when the "cut" menu item is selected
;       Pass:    nothing
;       Returns: nothing
;       Alters:
;
;*************************************************************************
DoCut:
        ;add your own code here
        rts

;*************************************************************************
;                       DoCopy
;
;       This is a dummy event handler routine. Customize this for your
;       own application.
;
;       Author:  Eric E. Del Sesto, August 1987
;       Caller:  R_DoCopy when the "copy" menu item is selected
;       Pass:    nothing
;       Returns: nothing
;       Alters:
;
;*************************************************************************
DoCopy:
        ;add your own code here
        rts

;*************************************************************************
;                       DoPaste
;
;       This is a dummy event handler routine. Customize this for your
;       own application.
;
;       Author:  Eric E. Del Sesto, August 1987
;       Caller:  R_DoPaste when the "paste" menu item is selected
;       Pass:    nothing
;       Returns: nothing
;       Alters:
;
;*************************************************************************
DoPaste:
        ;add your own code here
        rts
```

```
;*************************************************************************
;                       DoIcon1
;
;       This is a dummy event handler routine. Customize this for your
;       own application.
;
;       Author:  Eric E. Del Sesto, August 1987
;       Caller:  R_DoIcon1 when the "ICON" is pressed.
;       Pass:    nothing
;       Returns: nothing
;       Alters:
;
;*************************************************************************
DoIcon1:
        ;add your own code here
        rts

;*************************************************************************
;                       Edit Module local variables
;
;       This area contains definitions for variables which are local to
;       the Edit Module. No other module (including resident) can access
;       these variables. These variables are trashed whenever the Edit
;       Module swaps in or out, and so cannot be used for anything more
;       than temporary storage for routines in this module.
;
;*************************************************************************
        .ramsect
                                        ;variable section starts here
                                        ;(GeoLinker will give this an address
                                        ;of SWAP_VARS, which is $1f00.)
editVars:
        .block  1                       ;unused variable: for example only
```

```
;*************************************************************************
;
;                       SamVlirHdr
;
;       This file contains the header block definition for the GeoProgrammer
;       package sample VLIR application.
;
;Copyright (c) 1987 Berkeley Softworks. For the sole use of registered
;GeoProgrammer owners.
;*************************************************************************

.if     Pass1                   ;Only need to include this file
.noeqin                         ;during assembler's first pass.
.include geosSym                ;get GEOS definitions
.eqin
.endif

;Here is our header. The SamVlir.lnk file will instruct the linker
;to attach it to our sample application.

        .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:

                                    Vlir

        .byte   $80 | USR       ;Commodore file type, with bit 7 set.
        .byte   APPLICATION     ;Geos file type
        .byte   VLIR            ;Geos file structure type
        .word   ResStart        ;start address of program (where to load to)
        .word   $3ff            ;usually end address, but only needed for
                                ;desk accessories.
        .word   ResStart        ;init address of program (where to JMP to)

        .byte   "SampleVlir  V1.0",0,0,0,$00
                                ;permanent filename: 12 characters,
                                ;followed by 4 character version number,
                                ;followed by 3 zeroes,
                                ;followed by 40/80 column flag.

        .byte   "Eric E. Del Sesto  ",0
                                ;twenty character author name

        ;end of header section which is checked for accuracy
        .block  160-117         ;skip 43 bytes...
        .byte   "This is the GeoProgrammer sample "
        .byte   "VLIR GEOS application.",0
        .endh
```

```
;*******************************************************************************
;
;                       SamVlirZPVars
;
;       This file contains zero-page ($0000-$00ff) global variable definitions
;       for the GeoProgrammer package sample VLIR application. It is
;       included into each module (including resident) so that when each
;       module assembles, it knows the absolute zero-page address of these
;       variables.
;
;Copyright (c) 1987 Berkeley Softworks. For the sole use of registered
;GeoProgrammer owners.
;*******************************************************************************

        .zsect  a2                      ;we are using the a2-a9 area ($0070-$007f)
                                        ;(see geosMemoryMap)

curModule:                              ;holds module number of currently loaded
        .block  1                       ;module. See InitSwap and SwapMod.

;WARNING: do not place more than 16 bytes worth of variables here!
;We are restricted to the a2 - a9 area...
```

```
;*************************************************************************
;
;                                SamVlirEquates
;
;This file contains global equate definitions for the GeoProgrammer
;package sample VLIR application.
;
;Copyright (c) 1987 Berkeley Softworks. For the sole use of registered
;GeoProgrammer owners.
;*************************************************************************

;Miscellaneous equates:

NUM_MODS        = 2         ;this application has 2 swap modules:
MOD_FILE        = 1         ;record number for file module
MOD_EDIT        = 2         ;record number for edit module

SWAP_BASE       = $1000     ;module code loads from $1000
SWAP_SIZE       = $0f00     ;to $1eff
SWAP_VARS       = $1f00     ;modules use $1f00-$1fff as local var. area

NUM_DA          = 7         ;Geos menu can list names of 7 desk accessories

;Equates for jump tables in modules:

;File module:

J_RunDA         = SWAP_BASE + (0*3)  ;first entry in module's jump table
J_DoClose       = SWAP_BASE + (1*3)  ;2nd

;Edit module:

J_DoCut         = SWAP_BASE + (0*3)
J_DoCopy        = SWAP_BASE + (1*3)
J_DoPaste       = SWAP_BASE + (2*3)
J_DoIcon1       = SWAP_BASE + (3*3)
```

```
;Equates for main menu:

MM_COUNT        = 3         ;number of main menu items
MM_TOP          = 0         ;top scanline of menu
MM_BOTTOM       = 14        ;bottom scanline of menu
MM_LEFT         = 0         ;left pixel position of menu
MM_RIGHT        = 72        ;right pixel position of menu

SM_TOP          = MM_BOTTOM+1
;top of all sub-menus

;Equates for GEOS menu:

GM_COUNT        = 1         ;number of items (assuming no desk accessories-
                            ;InitDA routine will adjust table.)
GM_LEFT         = 0         ;left x position
GM_WIDTH        = 79        ;width in pixels

;Equates for FILE menu:

FM_COUNT        = 2         ;number of items
FM_LEFT         = 29        ;left x position
FM_WIDTH        = 40        ;width in pixels

;Equates for EDIT menu:

EM_COUNT        = 3         ;number of items
EM_LEFT         = 49        ;left x position
EM_WIDTH        = 40        ;width in pixels
```

```
;*************************************************************************
;
;                           SamVlir.lnk
;
;       This is the GeoLinker command file for the GeoProgrammer package
;       sample VLIR application.
;
;Copyright (c) 1987 Berkeley Softworks. For the sole use of registered
;GeoProgrammer owners.
;*************************************************************************

.output   SampleVlir        ;name for output file
.header   SamVlirHdr.rel    ;name of file containing header block to use

.vlir                       ;this is a VLIR application, resident module:

.psect    $0400             ;program code starts at $0400
.ramsect  $5000             ;global variable area starts at $5000

SamVlirRes.rel              ;name of file which contains relocatable
                            ;code and data from GeoAssembler

.mod      MOD_FILE          ;module 1: file module

.psect    SWAP_BASE         ;module swap code loads to $1000
.ramsect  SWAP_VARS         ;module local variable area

SamVlirFile.rel

.mod      MOD_EDIT          ;module 2: edit module

.psect    SWAP_BASE         ;module swap code loads to $1000
.ramsect  SWAP_VARS         ;module local variable area

SamVlirEdit.rel
```

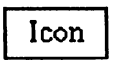
```
;*************************************************************************
;
;                       SamDA
;
;       This is the main file for the GeoProgrammer package sample
;       desk accessory. It contains all of the code and data required
;       for assembly.
;
;Note: A desk accessory:
;       -- should not alter the background screen area.
;       -- must honor the flag values passed from the application in r10L:
;       If B7=1, the DA must save the application's foreground screen
;       to a ram or disk buffer and restore it when returning to the app.
;       If B6=1, the DA must save and restore the application's color values
;       similarly.
;       -- must only use a specific, contiguous area of application memory
;       space (somewhere in $0400 to $5fff). The area used is specified
;       in the header block for the accessory. (See SamDAHdr.)
;       -- must fill its' screen section with the appropriate screen color.
;       It is a good idea to grab the color value from the card in the
;       top-right corner of the screen, so that your accessory's colors
;       will honor the Preference Manager settings.
;       -- must not use the top 16 scanlines of the screen.
;       -- must set its' own sprite picture data, colors, positions,
;       and X/Y doubling information.
;
;Since our accessory has menus, we always save and later restore the
;application's background screen space, so that we can use both the FG
;and BG screens, as a normal application would. Instead of saving the
;BG screen (FG screen and colors also if R10L dictates) to a temporary
;disk file, we save them in a big buffer which lies after the code in this
;file.
;
;Copyright (c) 1987 Berkeley Softworks. For the sole use of registered
;GeoProgrammer owners.
;*************************************************************************

.if     Pass1                   ;Only need to include these files
                                ;during assembler's first pass.
.include geosSym                ;get GEOS definitions
.include geosMac                ;get GEOS macro definitions
.endif
```

```
;Here are some equates to define our desk accessories' screen position.
;Everything is on card boundaries to simplify saving screen data and color
;information.

DA_TOP          = 8                 ;# of cards down from top of screen
                                    ;(MUST BE AT LEAST 2 FOR ALL DAs)
DA_LEFT         = 10                ;# of cards in from left side of screen
DA_HEIGHT       = 8                 ;# of cards high
DA_WIDTH        = 20                ;# of cards wide
                                    ;(MAX IS 32 or must rewrite SaveScreen and
                                    ;RestoreScreen routines.)
NUM_CARDS       = DA_HEIGHT*DA_WIDTH
                                    ;number of cards on screen covered by DA
FG_BUF_SIZE     = NUM_CARDS*8       ;size of buffer to save application's screen
                                    ;to. Equal to number of cards * 8 bytes/card.

;Our program starts here. The first thing we do is save the application's
;screen data and color information if necessary. Then we draw a box in the
;middle of the screen, initialize our menus and icons, and RTS to GEOS mainloop.
;When an event happens, such as the user selects a menu item or one of our
;icons, GEOS will call one of our handler routines.

        .psect                      ;program code section starts here
                                    ;(GeoLinker will give this an address of $1000)
DAStart:
                                    ;this label is needed when the linker
                                    ;resolves the SamDAHdr.rel file, so that the
                                    ;header block can have information about
                                    ;where to load the desk accessory.

        MoveB   r10L,recoverFlag    ;save flag passed from application

        jsr     SaveScreen          ;save application BG (and FG if necessary)
        jsr     SaveColors          ;save application colors if necessary
                                    ;and wipe with our color value

        LoadB   dispBufferOn,#(ST_WR_FORE | ST_WR_BACK)
                                    ;allow writes to FG and BG

        LoadW   r0,#DrawBox         ;point to graphics string to draw box
        jsr     GraphicsString

        LoadW   r0,#MenuTable       ;point to menu definition table
        lda     #0                  ;place cursor on first menu item when done
        jsr     DoMenu              ;have GEOS draw the menus on the screen

        LoadW   r0,#IconTable       ;point to icon definition table
        jsr     DoIcons             ;have GEOS draw the icons on the screen
        rts
```

```
;Here are some data tables for the init code shown above:

DrawBox:                                ;graphics string table to clear screen
        .byte   NEWPATTERN,0            ;set new pattern value (white)
        .byte   MOVEPENTO               ;move pen to:
        .word   DA_LEFT*8               ;top left corner of DB (in pixels)
        .byte   DA_TOP*8
        .byte   RECTANGLETO             ;draw filled rectangle to bottom right corner
        .word   (DA_LEFT+DA_WIDTH)*8 - 1
        .byte   (DA_TOP+DA_HEIGHT)*8 - 1
                                        ;bottom right corner of DB (in pixels)
        .byte   NEWPATTERN,1            ;set new pattern value (black)
        .byte   FRAME_RECTO             ;draw frame to...
        .word   DA_LEFT*8               ;top left corner of DB (in pixels)
        .byte   DA_TOP*8
        .byte   NULL

MenuTable:                              ;menu definition table for main horizontal menu
        .byte   DA_TOP*8                ;top y coordinate
        .byte   (DA_TOP*8)+14           ;bottom y coordinate
        .word   DA_LEFT*8               ;left x coordinate
        .word   (DA_LEFT*8)+44          ;right x coordinates
        .byte   2 | HORIZONTAL          ;number of menu items, type of menu

        .word   FileText                ;pointer to text for menu item
        .byte   VERTICAL                ;type of menu
        .word   FileSubMenu             ;pointer to menu structure

        .word   EditText                ;pointer to text for menu item
        .byte   VERTICAL                ;type of menu
        .word   EditSubMenu             ;pointer to menu structure

FileSubMenu:                            ;menu definition table for File vertical menu
        .byte   (DA_TOP*8)+15           ;top y coordinate
        .byte   (DA_TOP*8)+43           ;bottom y coordinate
        .word   DA_LEFT*8               ;left x coordinate
        .word   (DA_LEFT*8)+39          ;right x coordinates
        .byte   2 | VERTICAL            ;number of menu items, type of menu

        .word   CloseText               ;pointer to text for menu item
        .byte   MENU_ACTION             ;type of action
        .word   DoClose                 ;pointer to handler routine

        .word   QuitText                ;pointer to text for menu item
        .byte   MENU_ACTION             ;type of action
        .word   DoQuit                  ;pointer to handler routine
```

```
EditSubMenu:                            ;menu definition table for FILE vertical menu
        .byte   (DA_TOP*8)+15           ;top y coordinate
        .byte   (DA_TOP*8)+57           ;bottom y coordinate
        .word   (DA_LEFT*8)+20          ;left x coordinate
        .word   (DA_LEFT*8)+56          ;right x coordinates
        .byte   3 | VERTICAL            ;number of menu items, type of menu

        .word   CutText                 ;pointer to text for menu item
        .byte   MENU_ACTION             ;type of action
        .word   DoCut                   ;pointer to handler routine

        .word   CopyText                ;pointer to text for menu item
        .byte   MENU_ACTION             ;type of action
        .word   DoCopy                  ;pointer to handler routine

        .word   PasteText               ;pointer to text for menu item
        .byte   MENU_ACTION             ;type of action
        .word   DoPaste                 ;pointer to handler routine

;text strings for above menus

FileText:
        .byte   "file",0
CloseText:
        .byte   "close",0
QuitText:
        .byte   "quit",0
EditText:
        .byte   "edit",0
CutText:
        .byte   "cut",0
CopyText:
        .byte   "copy",0
PasteText:
        .byte   "paste",0
```

```
;icon definition table

IconTable:
        .byte   1               ;number of icons
        .word   DA_LEFT*8       ;x position to place mouse at when done
        .byte   DA_TOP*8        ;y position to place mouse at when done

        .word   Icon1Picture    ;pointer to compacted bitmap for icon
        .byte   DA_LEFT+3       ;x position in bytes
        .byte   (DA_TOP*8)+24   ;y position in scanlines
        .byte   ICON_1_WIDTH    ;width of icon in bytes
        .byte   ICON_1_HEIGHT   ;height of icon in scanlines
        .word   DoIcon1         ;pointer to handler routine

Icon1Picture:                   ;assembler will place compacted bitmap data
                                ;here for this picture:
```




```
ICON_1_WIDTH  = picW            ;store bitmap size values for use in above
ICON_1_HEIGHT = picH            ;table on pass 2. (picW and picH are set by
                                ;the assembler.)

;Event handler routines: are called by GEOS when an event happens,
;such as user selecting a menu item or clicking on an icon.

DoClose:
DoCut:
DoCopy:
DoPaste:
        jsr     GotoFirstMenu   ;roll menu back up

        ;code to handle this event goes here

        rts                     ;all done

DoQuit:
        jsr     GotoFirstMenu   ;roll menu back up
        jsr     RestoreColors   ;restore application's color values
        jsr     RestoreScreen   ;restore application's BG (and FG maybe) data
        jmp     RstrAppl        ;return to application!

DoIcon1:
        ;code to handle this event goes here
        rts
```

```
;*************************************************************************
;                   SaveScreen
;
;       This routine saves a portion of the application's FG and BG screens.
;
;       Author:  Eric E. Del Sesto, August 1987
;       Caller:  DAStart
;       Pass:    recoverFlag = r10L value passed from application
;       Returns: screenBuf = application's FG and BG screen data
;       Alters:  a, x, y, r0, r1, r5, r6
;
;*************************************************************************
SaveScreen:
        LoadW   r0,#screenBuf       ;use r0 as pointer to buffer
        ldx     #DA_TOP*8           ;use x as scanline #, start at top

10$:    ;for each card-row covered by DA. (Could make this a subroutine
        ;to save code space.)
        jsr     GetScanLine         ;get two pointers to screen data (r5 and r6)

        ;push two pointers to first byte in left-most card covered by DA
        AddVW   #(DA_LEFT*8),r5
        AddVW   #(DA_LEFT*8),r6

        ;start at right side of DA and read bytes to the left
        ldy     #(DA_WIDTH*8)-1 ;point to last byte in right-most card on line

20$:    ;for each byte in cards on this card-row
        lda     (r6),y          ;get byte from BG screen area
        jsr     SaveByte        ;and save it to buffer

        bit     recoverFlag     ;do we need to save FG also?
        bpl     30$             ;skip if not...

        lda     (r5),y          ;get byte from FG screen area
        jsr     SaveByte        ;and save it to buffer

30$:    ;on to next byte to the left
        dey
        cpy     #$ff            ;off left edge of DA space yet?
        bne     20$             ;loop for next byte if not...

        ;on to next card row
        txa                     ;add 8 (# lines per card) to scanline index
        clc
        adc     #8
        tax
        cpx     #(DA_TOP+DA_HEIGHT)*8
                                ;off bottom edge yet?
        bcc     10$             ;loop for next line if not...
        rts
```

```
;*************************************************************************
;                   RestoreScreen
;
;       This routine recovers a portion of the application's FG and BG screens.
;
;       Author:  Eric E. Del Sesto, August 1987
;       Caller:  DAStart
;       Pass:    recoverFlag = r10L value passed from application
;                screenBuf = application's FG and BG screen data
;       Returns: FG and BG screens restored
;       Alters:  a, x, y, r0, r1, r5, r6
;
;*************************************************************************
RestoreScreen:
        LoadW   r0,#screenBuf       ;use r0 as pointer to buffer
        ldx     #DA_TOP*8           ;use x as scanline #, start at top

10$:    ;for each card-row covered by DA
        jsr     GetScanLine         ;get two pointers to screen data (r5 and r6)

        ;push two pointers to first byte in left-most card covered by DA
        AddVW   #(DA_LEFT*8),r5
        AddVW   #(DA_LEFT*8),r6

        ;start at right side of DA and write bytes to the left
        ldy     #(DA_WIDTH*8)-1 ;point to last byte in right-most card on line

20$:    ;for each byte in cards on this card-row
        jsr     GetByte             ;get byte from buffer
        sta     (r6),y              ;and save to BG screen

        bit     recoverFlag         ;do we need to recover FG also?
        bpl     30$                 ;skip if not...

        jsr     GetByte             ;get byte from buffer
        sta     (r5),y              ;and save to FG screen

30$:    ;on to next byte to the left
        dey
        cpy     #$ff                ;off left edge of DA space yet?
        bne     20$                 ;loop for next byte if not...

        ;on to next card row
        txa                         ;add 8 (# lines per card) to scanline index
        clc
        adc     #8
        tax
        cpx     #(DA_TOP+DA_HEIGHT)*8
                                    ;off bottom edge yet?
        bcc     10$                 ;loop for next line if not...
        rts
```

```
;*************************************************************************
;                       SaveColors
;
;       This routine saves a portion of the application's color table.
;
;       Author:  Eric E. Del Sesto, August 1987
;       Caller:  DAStart
;       Pass:    recoverFlag = r10L value passed from application
;       Returns: colorBuf = application's color table data
;       Alters:  a, x, y, r0, r1, r2
;
;*************************************************************************
SaveColors:
        LoadW   r0,#colorBuf        ;use r0 as pointer to buffer
        LoadW   r2,#COLOR_MATRIX + (DA_TOP * 40) + DA_LEFT
                                    ;use r1 as pointer into color matrix
                                    ;storage area (start at top left of
                                    ;where DA lies).
        ldx     #DA_HEIGHT          ;use x as card-line counter

10$:    ;for each card-line covered by DA
        ;start at right side of DA and read bytes to the left

        ldy     #DA_WIDTH-1         ;point to right-most card on line

20$:    ;for each card on line: first save card color if necessary

        bit     recoverFlag         ;do we need to save application's colors?
        bvc     30$                 ;skip if not...

        lda     (r2),y              ;get byte from COLOR_MATRIX area
        jsr     SaveByte            ;and save it to buffer

30$:    ;and now stuff card with value we want

        lda     COLOR_MATRIX+40-1
                                    ;get card color value from top-right corner
                                    ;of application's screen
        sta     (r2),y              ;and use as color for this card

        ;on to next byte to the left

        dey                         ;off left edge of DA space yet?
        bpl     20$                 ;loop for next byte if not...

        ;on to next line

        AddVW   #40,r2              ;push pointer to next line in COLOR_MATRIX
        dex                         ;one less line to go
        bne     10$                 ;loop if more lines to go...
        rts
```

```
;*************************************************************************
;                   RestoreColors
;
;       This routine recovers a portion of the application's color table.
;
;       Author:  Eric E. Del Sesto, August 1987
;       Caller:  DAStart
;       Pass:    recoverFlag = r10L value passed from application
;                colorBuf = application's color table data
;       Returns: COLOR_MATRIX updated
;       Alters:  a, x, y, r0, r1, r2
;
;*************************************************************************
RestoreColors:
        bit     recoverFlag         ;do we need to recover application's colors?
        bvc     90$                 ;skip if not...

        LoadW   r0,#colorBuf        ;use r0 as pointer to buffer

        LoadW   r2,#COLOR_MATRIX + (DA_TOP * 40) + DA_LEFT
                                    ;use r1 as pointer into color matrix
                                    ;storage area (start at top left of
                                    ;where DA lies).

        ldx     #DA_HEIGHT          ;use x as card-line counter

10$:    ;for each card-line covered by DA
        ;start at right side of DA and stuff bytes to the left

        ldy     #DA_WIDTH-1         ;point to right-most card on line

20$:    ;for each card on line: restore card color value
        jsr     GetByte             ;get byte from buffer
        sta     (r2),y              ;save byte to COLOR_MATRIX area

        ;on to next byte to the left

        dey                         ;off left edge of DA space yet?
        bpl     20$                 ;loop for next byte if not...

        ;on to next line

        AddVW   #40,r2              ;push pointer to next line in COLOR_MATRIX
        dex                         ;one less line to go
        bne     10$                 ;loop if more lines to go...

90$:    ;all done
        rts
```

```
;*************************************************************************
;                       SaveByte, GetByte
;
;       These two routines are used to save/recall a byte to/from the
;       screen and color buffers.
;
;       Author:  Eric E. Del Sesto, August 1987
;       Caller:  SaveScreen, RestoreScreen, SaveColors, RestoreColors
;       Pass:    r0 = pointer into screenBuf or colorBuf
;                a = value to save (SaveByte)
;       Returns: r0 = pointer to next byte in buffer
;                a = value from buffer (GetByte)
;                x,y = same as before
;       Alters:  a, r1L
;
;*************************************************************************
SaveByte:
        sty     r1L             ;save y register temporarily
        ldy     #0
        sta     (r0),y          ;save byte into buffer
        bra     Finish          ;skip ahead to finish up...

GetByte:
        sty     r1L             ;save y register temporarily
        ldy     #0
        lda     (r0),y          ;get byte from buffer

Finish:
        inc     r0L             ;increment pointer (these three lines
        bne     90$             ;constitute the IncW macro.)
        inc     r0H
90$:    ldy     r1L             ;restore y register
        rts
```

```
;*****************************************************************************
;       Global Variables
;
;These variables are placed IMMEDIATELY following our DA code so that
;our entire DA (code+variables) is one contiguous block of memory.
;*****************************************************************************

        .ramsect
                                ;data storage area starts here

recoverFlag:
        .block  1               ;holds flags passed from application in r10L

screenBuf:
        .block  FG_BUF_SIZE*2   ;holds application's FG and BG screen data
                                ;while DA is running

colorBuf:
        .block  NUM_CARDS       ;holds application's card color info
                                ;while DA is running

DAEnd:
                                ;DA ends here. Linker needs this value
                                ;for SamDAHdr file.
```

```
;*************************************************************************
;
;                       SamDAHdr
;
;       This file contains the header block definition for the GeoProgrammer
;       package sample desk accessory.
;
;Copyright (c) 1987 Berkeley Softworks. For the sole use of registered
;GeoProgrammer owners.
;*************************************************************************

.if     Pass1                   ;Only need to include this file
.noeqin                         ;during assembler's first pass.
.include geosSym                ;get GEOS definitions
.eqin
.endif

;Here is our header. The SamDA.lnk file will instruct the linker
;to attach it to our sample desk accessory.

        .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:

                                        DA

        .byte   $80 | USR       ;Commodore file type, with bit 7 set.
        .byte   DESK_ACC        ;Geos file type
        .byte   SEQUENTIAL      ;Geos file structure type
        .word   DAStart         ;start address of program (where to load to)
        .word   DAEnd           ;end address (VERY IMPORTANT)
        .word   DAStart         ;init address of program (where to JMP to)
        .byte   "SampleDA   V1.0",0,0,0,$00
                                ;permanent filename: 12 characters,
                                ;followed by 4 character version number,
                                ;followed by 3 zeroes,
                                ;followed by 40/80 column flag.
        .byte   "Eric E. Del Sesto ",0
                                ;twenty character author name

        ;end of header section which is checked for accuracy
        .block  160-117         ;skip 43 bytes...
        .byte   "This is the GeoProgrammer sample "
        .byte   "GEOS desk accessory.",0
        .endh
```

```
;*************************************************************************
;
;                           SamDA.lnk
;
;       This is the GeoLinker command file for the GeoProgrammer package
;       sample desk accessory.
;
;Copyright (c) 1987 Berkeley Softworks. For the sole use of registered
;GeoProgrammer owners.
;*************************************************************************

.output   SampleDA        ;name for output file
.header   SamDAHdr.rel    ;name of file containing header block to use

.seq                      ;this is a sequential application

.psect    $1000           ;program code starts at $1000 (label called
                          ;(DAStart will get this value.)
                          ;start ram section immediately following program code.

SamDA.rel                 ;name of file which contains relocatable
                          ;code and data from GeoAssembler
```

Appendix B: geoProgrammer File Formats.

.rel File Format

The relocatable object file output from geoAssembler, and used by geoLinker, is a VLIR file with four records:

record 0:
relocatable 6502 machine language — the actual assembled 6502 code with zeros as placeholders for unresolved expressions. All relocatable references (references which were resolved during the assembly) are relative offsets from the first byte of the module. To relocate these relative expressions the linker uses the information in record 2.

record 1:
For each unresolvable expression, the following exist:
— expression text string terminated with a null byte ($00).
— two byte (low/high) pointer into the relocatable object code.
— one byte length count. The length count is the length of the whole instruction, which is always one more than the actual number of bytes to store. 2 = 1 byte and 3 = 2 bytes.

record 2:
relocation table — a table of two-byte (low/high) pointers into record 0. Each entry points to a relocatable address. The linker walks this list and adds the appropriate base address to the word values pointed to in the relocatable object code of record 0. This is the relocation process.

record 3:
psect size, ramsect size, and symbol table:
bytes 0,1: size of psect section (low/high)
bytes 2,3: size of ramsect section (low/high)
remainder of file: symbols, 10 bytes for each — only the first eight characters of a symbol name are used during assembly and linking. If a symbol is less than eight characters, the rest will be padded with spaces.
bytes 0-7 eight character symbol text, padded with zeros if less
bytes 8-9 symbol value in low/high order
— bit 7 (MSB) of the first four characters in the symbol name are used as flags:

a. The MSB of the first character is set if it is a psect label and its address should be relocated during linking.

b. The MSB of the second character is set if it is a ramsect label and its address should be relocated during linking.

c. The MSB of the third character is set if it is a zsect label (this flag is not used by the linker).

d. The MSB of the fourth character is set if the symbol is an equate defined with the = (single equal sign) assembler directive. This symbol will not be written to the .dbg debugger symbol table.

.dbg File Format

The debugger symbol table file output from geoLinker, and used by geoDebugger, is a VLIR file with a variable number of records. Each record number corresponds to the appropriate module number in the file. For sequential applications, only record zero is used. For VLIR applications, record zero contains the symbols for the resident module and the other records contain symbols for the application's overlay modules. Symbols in each module are sorted numerically.

Each symbol uses ten bytes:

bytes 0-7 eight character symbol text, padded with spaces if less than eight.

bytes 8-9 symbol value in high/low order (note: this is different from the low/high order of the .rel file).

Appendix C: geoDebugger Technical Notes

Super-debugger Primitives

All super-debugger commands are built from one or more command primitives. These command primitives are comprised of an @ symbol followed by another character. Command primitives generally execute faster than their system macro equivalents and can be used in user-defined macros. Refer to "Macro Commands" in Chapter 8 for more information.

primitive	*function*
@'	**w**
@(	**getb**
@)	**putb**
@*	**drivea**
@+	**driveb**
@,	**dumpd**
@-	**disk**
@.	**quit**
@/	**a**
@0	**poff**
@1	**setu**
@2	set u.fn (used by **for** command)
@3	do for loop (used by **for** command)
@4	**stop**
@5	**mod**
@6	**setmod**
@7	**initmod**
@8	**m**
@9	**sysmac**
@:	**initmac**
@;	**clrmac**
@<	**setmac**
@=	**mac**
@>	**pc**
@?	**if**
@@	**opt**

primitive	*function*
@a	**rboot**
@b	**b**
@c	**setb**
@d	**clrb**
@e	**initb**
@f	**find**
@g	**go**
@h	**pon**
@i	**dump**
@j	**initsym**
@k	**clrsym**
@l	**sym**
@m	**copy**
@n	**setsym**
@o	**stopmain**
@p	**p**
@q	**jsr**
@r	**r**
@s	**s**
@t	**t**
@u	**finish**
@v	**diff**
@w	**inithist**
@x	**print**
@y	**hist**
@z	**fill**
@[	**reg**
@]	**flag**
@^	**runto**
@_	**next**

Startup Conditions

When geoDebugger loads an application for debugging, it prepares the environment by doing the following:

1: The entire program memory space is cleared:
   $400-$5fff (super-debugger)
   $400-$3dff (mini-debugger)

Note: if your application runs fine from the debugger but not from the deskTop it could be that you are forgetting to initialize some variables, assuming they will be zero. Because the deskTop does not clear the program space, the ramsect regions may contain random values.

2: All GEOS registers (**r0** through **r15**) except **r10L** are cleared to zero. **r10L** is loaded with $c0, which signals a "worst-case" situation for a desk accessory (**FG_SAVE** and **CLR_SAVE** bits both set). If your desk accessory operates correctly under these conditions, it should work fine under others.

Debugger Isolation

geoDebugger isolates itself almost completely from the application and GEOS. However, the disk-related commands require it to use of **SetDevice**, **OpenDisk**, **GetBlock**, and **PutBlock**. The debugger is otherwise entirely self-contained.

Off Limits Memory

At no time should the application modify the following memory areas:

1: $350-3ff (debugger kernal)
2: $314-$319 (BASIC interrupt vectors)
3: $fffa-$ffff (interrupt vectors)

Protected Memory

While in the debugger, the following memory locations cannot be altered. If they are viewed, they will always be displayed as $ee regardless of their actual values.

$0100 to stack ptr.	memory below stack pointer is used by debugger.
$0314 to $0319	BASIC interrupt vectors.
$0350 to $03ff	debugger kernal area.

$fffa to $ffff interrupt vectors.

In the mini-debugger, the following area is also off limits:

$3e00 to $5fff mini-debugger.

Miscellaneous

geoDebugger sets a **brk** instruction over the **EnterDeskTop** vector in case the application calls **EnterDeskTop**. The only safe way to disable the debugger and return to the deskTop is with the **quit (q)** command.

With the ROM bank enabled with the MM register, any attempt to modify the ROM memory area will actually modify the RAM which it is swapped over. This is why attempting to set a breakpoint in ROM will store a $00 (a **brk** instruction) in the RAM it maps over.

The application can modify the memory map register (location $0001) directly. geoDebugger will sense the state and adjust appropriately.

The geoDebugger **quit** command assumes that GEOS and the reserved areas of zero-page are intact. If they are not, the system may crash, requiring a complete power-down to reset. With the super-debugger, the **rboot** command can be used if you suspect that parts of GEOS or zero-page have been destroyed.

Appendix D: Bibliography and Further Reference

6502 Assembly Language

Programming the 6502, **Rodnay Zaks, SYBEX Computer books, 2344 Sixth Street, Berkeley, CA 94710 (1983).** One of the most lucid and well-liked introductory books on 6502 assembly language.

6502 Software Design, **Leo J. Scanlon, Howard W. Sams & Co., Inc., 4300 West 62nd Street, Indianapolis, IN 46268 (1980).** Another good introductory 6502 assembly language text; places special emphasis on the more advanced aspects of 6502 programming and includes working subroutines for base-conversion, lookup tables, and math operations.

MCS6500 Microcomputer Family Programming Manual, **MOS Technology, Faulk Baker Associates.** The official guide to the 6502 (and descendents); useful but not essential because all the aspects in this book are covered by other 6502 books.

Commodore 64

Commodore 64 Programmer's Reference Guide, **Commodore Business Machines, Inc., Computer Systems Division, 487 Devon Park Drive, Wayne, PA 19807 (1982).** Contains useful information describing the Commodore 64 environment, such as the memory map register, the display controller, and the sound controller.

GEOS

The Official GEOS Programmer's Reference Guide, **Berkeley Softworks, Bantam Books, Inc., 666 Fifth Avenue, New York, NY 10103 (1987).** The essential book for programming under the GEOS environment. Covers all GEOS file formats, structures, routines, and calling conventions in detail.

Appendix E: Error Messages

This appendix is divided into three sections: disk related errors, geoAssembler errors, and geoLinker errors. Disk related errors are identical in both geoAssembler and geoLinker and are displayed in a dialog box; other errors are specific to either geoAssembler or geoLinker and are written into the .err file. A *fatal error* is an error which aborts the assembly or link.

Disk Related Errors

If geoAssembler or geoLinker encounters a disk error, the error will be displayed in a dialog box. Disk errors are always fatal.

Disk full
There was insufficient room on the disk to complete the attempted write operation. Delete unused files from the disk or spread your files across two disk drives to free-up space.

File Not Found
The program was unable to find a file. A common error is a typo in the file name or trying to use a file with a space character somewhere in its name.

Drive Not Responding
The disk drive is not responding to the read or write request.

Bad disk/no disk
The disk is completely unreadable or there is no disk in the drive.

Disk write error
Either the write verify failed or an invalid track error occurred. In either case, this usually means a bad disk.

Disk write protected
The disk drive is unable to write to the disk because the write-protect notch is covered.

Disk name mismatch
The expected disk was not in the drive; usually means the user swapped disks when he should not have.

Bad allocation map
This disk block allocation map (BAM) contains bad values; usually indicates a destroyed disk.

General disk error
All other disk errors — shows the actual GEOS disk error number.

geoAssembler Errors

Hidden error found
An error was detected on the first pass of the assembler but not on the second pass. Remove the **Pass1** conditional and reassemble. The errors will be flagged with complete error messages. When you have corrected the errors, you can replace the **Pass1** conditional.

Parse buffer overflow
The line is too complicated to fit into the parse buffer. Simplify the line or break it into several parts. Possibly fatal.

Missing parameter
This directive requires a parameter.

Branch to an external address
The destination of a branch instruction cannot make an external reference. A common cause of this error is a mistyped label name, which geoAssembler interprets as an external reference.

Invalid local label
A bad character was found in a local label definition; too many characters in a local label.

Multiple definition of a local label
You tried to use the same local label twice in the same local region (the area between two successive global labels). This error can also be caused by a macro-generated local label: when macros expand their internal labels are converted to local labels which count backward from 9999$. A high-number local label might conflict with one of these.

Too many local labels
You cannot define more than 20 local labels in the same local region (the area between two successive global labels). This error can also be caused if macro-generated local labels push the total over 20.

Illegal character in symbol
Symbols must begin with a letter or an underscore and may only contain letters, numbers, and underscore characters.

Label too long
You tried to define a label longer than 20 characters.

Multiple definition of a global label
The same global label was defined more than once.

Symbol table full
Too many symbols were defined. Fatal error.

Illegal addressing mode
The operand supplied is not a valid addressing mode for this 6502 instruction. This can also be caused by using a local label in the wrong context (anywhere except as the destination of a branch instruction).

Unknown opcode
A non-existent 6502 mnemonic, macro, or directive was found in the opcode field.

No .ENDH found for .HEADER
Every **.header** must have a matching **.endh.** Fatal error.

Macro label table overflow
Too many macro labels were defined. If macros are being nested, this is a cumulative error generated by labels defined in the macros nested in this invocation. Note: this tabel is only used for macro labels which are converted to local labels (labels which are not passed as parameters); labels which are passed as parameters don't have this limitation.

Macro parameter overflow
Too many macro parameters were used. This is a cumulative error generated by the combined length of all the parameters defined in the macros nested in this invocation.

Zsect overflow
The zsect counter exceeded page zero ($ff) as the result of a **.block**, or the parameter to a **.zsect** directive was greater than $ff. Fatal error.

Expression must evaluate fully when encountered
This expression cannot contain external or forward references because it must evaluate to an absolute number during the first pass of the assembly. Possibly fatal.

Missing file name
An expected file name was not found (**.include**).

Byte expression greater than $ff
A word value was found where a byte expression was expected; the low byte was used. Use the [low-byte operator to avoid this warning.

.IF nesting error
.if constructs cannot be nested deeper than ten levels. Fatal error.

.ENDIF found without .IF
A **.endif** was found without a matching **.if**.

No .ENDIF for .IF
A **.if** was found without a matching **.endif**. Fatal error.

More than one .ELSE statement
Each **.if** can only have one corresponding **.else** directive.

.ELIF statement after .ELSE
An **.elif** cannot follow an **.else**, only a **.if** or another **.elif**.

Branch out of range
Branch instructions have range of -127 to +128 bytes. The attempted branch exceeded this range.

Malformed expression
The expression evaluator was unable to parse and interpret the expression. Common causes of this error are mismatched parentheses or invalid operators.

Missing macro name
A **.macro** was found without a macro name.

Too many macro parameters
A macro definition cannot specify more than six parameters.

Macro already defined, definition ignored
A warning that a macro was defined more than once. Only the first macro definition is acknowledged.

Invalid macro parameter
A parameter declaration in the macro definition has an invalid character in it.

.INCLUDE nesting overflow
Include files may only be nested to a level of three. Fatal error.

Macros nested too deep
Macros can only be nested to a level of three.

Bad character string
A character string is missing a closing quote or has bad characters after the closing quote.

No .ENDM found for .MACRO
A **.macro** directive must have a matching **.endm.** Fatal error.

.ELSE found without .IF
An **.else** should have a matching **.if.**

.ELIF found without .IF
An **.elif** should have a matching **.if.**

Undefined local label
A undefined local label was referenced.

Macro name too long
A macro name may not exceed twenty characters.

Macro parameter name too long
A macro parameter name cannot exceed ten chracters.

Illegal character in macro name
Macro names must begin with a letter or an underscore and may only contain letters, numbers, and underscore characters.

Bitmap data not allowed in a macro definition
You cannot have a bitmap image inside a macro.

Too many macro definitions
There is no more room left in the macro table for this macro. Fatal error.

Macro text buffer overflow
The total text size of all defined macros is too large. Either shorten or remove macros. Fatal error.

No parameter is allowed for .psect
Psect sections are always relocated during the link stage and so cannot given an address during assembly.

Cannot use relocatable label as a parameter
A relocatable label was used as a parameter to a directive.

Inappropriate context for directive
This directive cannot be used in this context. For example: **.macro** within a header definition.

In file header
The automatic checking in the header definition found a mismatch. Make sure you are using the proper directive (**.byte/.word**) with the proper number of bytes.

Line too long
A geoAssembler source line cannot exceed 140 characters.

Expression too complex
The expression has too many operators or the parentheses are nested too deeply. Simplify it or break it up into subexpressions.

Object code too large
geoAssembler cannot generate a .rel module with more than about 6K of object code.

Too many errors
geoAssembler found more than 99 errors during this assembly. Fatal error.

geoLinker Errors

Parse buffer overflow
The line is too complicated to fit into the parse buffer. Simplify the line or break it into several parts.

Illegal module number
A module number specified in a **.mod** directive must be in the range 1<=n<=126.

Module already exists
This module number has already been used in a previous **.mod** directive.

Resident symbol table overflow
Too many symbols in the resident module. Use the geoAssembler **.noglbl** and **.noeqin** directives to reduce the number of symbols sent to the linker.

Overlay module symbol table overflow
Too many symbols in an overlay module. Use the geoAssembler **.noglbl** and **.noeqin** directives to reduce the number of symbols sent to the linker.

Overlay module not allowed for SEQ or CBM applications
A **.mod** directive was found after a **.seq** or a **.cbm** directive.

Missing or unresolvable argument
The argument in this directive cannot be resolved or is missing.

End of file encountered prematurely
geoLinker expected more information in the linker command file.

Expression cannot be resolved
An external reference was not resolved.

File name expected
geoLinker was expecting to find a file name here.

More than one page in .lnk file
The linker command file cannot exceed one geoWrite page.

Unknown directive or inappropriate context for directive
This directive does not exist or cannot be used here.

Picture data not allowed in command file
geoLinker found a bitmap pasted in the command file.

Resident module cannot start with .mod
A **.vlir** was expected.

Missing file name
This geoLinker directive requires a filename as a parameter.

Symbol defined more than once
A symbol in this module which exists in more than one .rel file was referenced.

Header file not exactly 256 bytes
The file specified in the **.header** directive is expected to contain exactly 256 bytes of object code.

Too many overlay modules
geoLinker cannot handle more than 20 overlay modules.

Bad syntax on line
This line is malformed and does not match the linker command file sequence. Usually results from a comment with a missing semicolon or garbage at the end of a line.

More than one .psect found
Each module can only have one **.psect.**

More than one .ramsect found
Each module can only have one **.ramsect.**

Too many symbols for .sym file
There are too many symbols for the .sym viewable symbol file. Use the geoAssembler **.noglbl** and **.noeqin** directives to reduce the number of symbols sent to the linker.

Header directive not allowed for .cbm file
You cannot use the **.header** directive when generating a standard Commodore (CBM) type application. geoLinker will generate the CBM header automatically.

Line too long
This geoWrite text line in the linker command file is too long for the geoLinker line buffer.

Page buffer overflow
The linker command file file cannot exceed one geoWrite page and this page is of a limited size. Try removing some of the comments from the linker command file.

Glossary

6502	A microprocessor developed in the mid-1970s by MOS Technology; the Commodore 64 and 128 use the 6510 and 8502 microprocessor, respectively, both of which are software-compatible with the 6502. geoAssembler accepts 6502 assembly language source code.
A	Accumulator. The 6502's general purpose register.
absolute address	A specific memory address ($0000-$ffff) in the Commodore's memory space. Program and memory spaces get assigned to absolute addresses by geoLinker. Compare with *relocatable address.*
address	A memory location. Possible addresses in the Commodore 64 range from $0000 to $ffff. 6502 addresses are stored in low/high order.
addressing mode	The 6502 has eight different addressing modes; specified in the operand of the instruction, they determine how values and memory locations are referenced by the instruction.
alphanumeric	Referring to ASCII letters and numbers.
application	A runnable program. A *GEOS Application* is a program designed to run in, and take advantage of, the GEOS environment.
arithmetic expression	An expression which uses arithmetic operators and evaluates to a 16-bit value. Compare with *logical expression.*
argument	A parameter used in a directive or a macro invocation
assembler	The program which converts assembly language source code into machine language or relocatable object code.

assembly language	The combination of 6502 mnemonics, operands, labels, comments, and directives used as the source input to the assembler. An assembly language program must first be assembled (and linked) before it can be run. Compare with *machine language*.
B	Break flag. Bit four in the 6502 status register. If set, indicates a **brk** instruction was encountered.
backward reference	In assembly language, a reference to a symbol in the current assembly which is previously defined.
binary	The base-two numbering system which consists of 0's and 1's. The binary radix symbol in geoProgrammer is **%**.
bit	An individual *b*inary dig*it* in a byte. Can be either 1 (set) or 0 (clear).
bitmap	A graphic image. Bitmaps can be pasted into your geoAssembler source code. See also *compacted bitmap*.
branch	A type of 6502 instruction which jumps to a new memory location, relative to the current location, based on the bits in the status register.
breakpoint	In geoDebugger, the location of an instruction in your program which can be set so that the debugger will be entered when the instruction is encountered but before it is executed. See also *conditional breakpoint*.
bug	A problem, mistake, or malfunction in a program.
byte	The basic unit of memory used by the 6502. Each memory location holds a unique byte. A byte consists of eight bits (numbered 0-7, right to left) and can range from 0-255 ($00-$ff) for unsigned numbers or +127 to -128 for signed, two's-complement numbers.

C	Carry flag. Bit zero in the 6502 status register. If set, indicates a carry from an arithmetic instruction.
call	To execute a routine, usually with a **jsr** instruction.
case dependency	Whether or not letter-case (upper/lower) is significant. As a general rule, mnemonics, directives, and hexadecimal numbers may be typed in upper- or lower-case, or some mixture thereof, and they will be interpreted identically, whereas each different upper/lower-case combination in a label, equate, or macro name will be considered unique.
code field	On a geoAssembler source code line, the field which contains the 6502 instruction, macro invocation, or directive. The code field is broken down into the *opcode field* and *operand field.*
comment	An explanatory note or text within your source code, linker command file, or debugger macro file. It is analagous to the BASIC **REM** statement. Comments are preceded by a **;** (semicolon) character.
comment field	On a geoAssembler source line, the field which contains the comment.
Commodore application	A non-GEOS program.
compacted bitmap	A bitmap stored in a special compressed format. GEOS contains routines for decoding compacted bitmaps. Bitmaps pasted into geoAssembler source code are converted to compacted bitmap data during assembly.
conditional assembly	Use of the **.if** family of assembler directives to include or disinclude source lines based on the result of a logical expression.

conditional breakpoint	A geoDebugger breakpoint which will only succeed if a specified logical expression evaluates to true.
constants file	A geoAssembler include file which consists entirely of equated constants
cross-reference	A situation where two independently assembled source code modules make external references to each other.
cross-assembler	An assembler which runs on one machine, but generates programs for another machine with a (usually) very different architecture. geoAssembler is based on Berkeley Softworks' in-house cross-assembler.
D	Decimal mode flag. Bit three in the 6502 status register. If set, indicates that arithmetic instructions will be performed in decimal mode. Be sure the decimal flag is in a known state before performing arithmetic instructions, especially in interrupt code.
.dbg	File name extender for a debugger symbol file.
.dbm	File name extender for a debugger macro file.
debugger	A tool for tracking down and eliminating programming bugs.
debugger isolation	The degree to which a debugger isolates itself from the application being debugged. The higher the degree of isolation, the more transparent and impervious the debugger. geoDebugger is extremely isolated.
debugger screen	One of two screen displays in geoDebugger. The debugger screen is a text display which chronicles your interaction with the debugger. Compare with *GEOS screen.*
decimal	The base-ten numbering system which uses the symbols 0-9. Decimal is the default radix in

	geoAssembler and, therefore, has no radix character there. In geoDebugger, however, a . (period) indicates a decimal number.
desk accessory	A sequential application designed to be accessible from the **geos** menu.
directive	A geoAssembler command which appears in the code field and is usually preceded by a period character. Also called *pseudo-op*.
disassemble	In geoDebugger, the process of converting machine language bytes into standard 6502 mnemonic plus addressing mode form.
equate	An explicit definition of a symbol in geoAssembler using the = or == directive. Equates can be absolute addresses or constants.
.err	geoAssembler and geoLinker error file extender.
event	Some sort of occurrence, such as a keypress, a mouse click, a menu selection, or a timer countdown, which GEOS recognizes and calls an application's event routine as a result.
event-driven program	An application which is centered around waiting for events. GEOS applications are event-driven.
executable file	An application which can be run. Also called *runnable file*.
expression	Any valid combinations of symbols, numeric constants, and operators which the expression evaluator recognizes.
expression evaluator	A routine which parses, interprets, and evaluates expressions.

external reference — In geoAssembler, a reference to a symbol which exists in an independently assembled file. The reference will be resolved by geoLinker.

false — A logical truth-value: if an expresion is false, it evaluates to an arithmetic zero ($0000); an arithmetic zero ($0000) is considered false. Compare with *true*.

firewalling — A debugging technique where routines are isolated from each other and interact only by affecting a group of variables. This limits the possibility of one routine corrupting another.

forward reference — In assembly language, a reference to a symbol defined later in the source file.

GEOS equate — An official equate for use with GEOS as defined in the **geosConstants**, **geosRoutines**, and **geosMemoryMap** sample include files.

geoWrite document — A text file compatible with geoWrite. geoProgrammer source code, .err, .lnk, .sym, and .dbm files are all geoWrite documents.

global label — A normal label which can be referenced anywhere in the current assembly and, unless suppressed with the **.noglbl** directive, can also be referenced externally.

header — A GEOS file header; contains information about the program, including the deskTop icon image.

hexadecimal — The base-sixteen numbering system which consists of the characters 0-9 and a-f. The hexadecimal radix symbol in geoProgrammer is $. Hexadecimal is sometimes referred to as simply "hex."

high-byte — In a two-byte number, the most-significant byte. See also *word*, *low/high order*, *low-byte*.

hot key — In geoDebugger, pressing RESTORE while your application is running will interrupt the processor and pass control to geoDebugger.

I	Interrupt disable flag. Bit two in the 6502 status register. If set, IRQ interrupts will be disabled.
include file	A geoAssembler source file designed to be used with the **.include** directive.
index register	Either the 6502 X or Y register.
initialized data	In geoAssembler, data defined in a psect section with **.byte** or **.word** (or **.block**), the data is said to be initialized because actual values are generated in the object code. Compare with *unitialized data.*
label field	On a geoAssembler source code line, the field which contains the label.
linker command file	The .lnk file which tells geoLinker how to create the runnable application.
.lnk	file name extender for the linker command file.
local label	In geoAssembler, a label which is local to the area between two successive global labels. Local labels are a one to four digit number followed by a dollar sign (**nnnn$**). Local labels can only be used as the destination of a branch instruction.
location counter	In geoAssembler, a counter which keeps track of the current object code position; geoAssembler has three location counters — one for each of zsect, psect, and ramsect.
logical expression	An expression which uses logical operators and evaluates to either true or false. Compare with *arithmetic expression.*
low-byte	In a two-byte number, the least-significant byte. See also *word, low/high order, low-byte.*
low/high order	A 6502 convention where two-byte numbers (such as addresses) are stored with the high-byte following the low-byte.

LSB	Least Significant Bit — bit zero; Least Significant Byte — low-byte.
machine language	The raw, numeric representation of a program which is run by the 6502. The assemble and link process converts assembly language into machine language.
macro	In geoAssembler, a set of source lines assigned to a name so that they may be inserted anywhere in the code by referring to the name; In geoDebugger, a set of keystrokes and commands assigned to a name so that they may be executed by referring to the name.
macro assembler	An assembler, like geoAssembler, which implements a macro facility.
macro definition	The source lines which describe a macro and its parameters. Compare with *macro invocation.*
macro expansion	When invoking a macro, the parameters are substituted into the source lines and the macro is placed directly into the input stream. The invocation is said to "expand" into the full-size of the macro.
macro invocation	Using a macro. Compare with *macro definition.*
microPORT	The Berkeley Softworks development system (cross-assembler, cross-linker, and in-circuit emulator) upon which geoProgrammer is based.
MM	Memory Map register. A register which contains the Commodore 64 memory map (switchable memory bank) status.
modular programming	A programming technique where the application is broken down into multiple source files called modules.
module	In an application, a small source file (among many) which contains routines of similar nature. The breakdown of the source code into modules is

	conceptually useful, but not necessary. Sometimes used to refer to *overlay module*.
MSB	Most Significant Bit — The high-order bit (bit 7 in a byte); Most Significant Byte — the high-byte in a word.
N	Negative flag. Bit seven in the 6502 status register. If set, the arithmetic result was negative.
nesting	The process of using a construct inside of itself. For example: calling a macro from within a macro, using a conditional inside of a conditions, or including a file from within an include file.
object code	As in "relocatable object code," the .rel output from geoAssembler.
octal	The base-eight numbering system which consists of the characters 0-7. The octal radix symbol in geoProgrammer is **?**.
one's complement	The bit-by-bit binary negation of a number, where all one's become zeros and all zeros become ones. In the *one's complement numbering system*, a negative number is the one's complement of its positive counterpart. Compare with *two's complement*.
opcode	A 6502 instruction.
opcode field	In geoAssembler, a subfield of the code field which holds the 6502 mnemonic.
operand	A 6502 addressing mode or value; the opcode "operates" with the operand.
operand field	In geoAssembler, a subfield of the code field which holds the 6502 operand.
operator	Characters, such as + or / which cause the expression evaluator to perform some action on one or two subexpressions. A *unary* operator works with

	one subexpression; a *binary* operator works with two.
operator precedence	The priority table which the expression evaluator uses to determine which operations to perform first in a complex expression.
overlay linker	A linker, like geoLinker, which supports overlay modules.
overlay module	A record in a VLIR file which contains machine code designed to be loaded into memory as needed, overlaying code which is no longer needed.
page	In 6502 memory space, a group of 256 bytes beginning on a 256-byte boundary. Page 0 is the first 256 bytes of memory ($0000-$00ff), page 1 is the second 256 bytes ($0100-$01ff), and so on.
parameter	An argument or value used with a macro or a directive.
parser	A routine which interprets a string of commands or expressions, breaking it down into its syntactic elements. The routine is said to "parse" the string.
pass	In geoAssembler, reading through and interpreting an entire source file once. geoAssembler makes two passes on the source file in order to generate the relocatable object file.
Pass1	In geoAssembler, an internal variable which evaluates to logical true on the first pass and logical false on the second; useful for avoiding a redundant pass on equate and macro include files.
patching	In geoDebugger, to use the **a** (assembly) mode to modify and test your program.
PC	Program Counter register. The two-byte 6502 register which points to the next instruction in memory to execute.

phase error	An assembler condition which geoAssembler cannot recognize; occurs when variables and equates evaluate to different values on each pass. Only occurs when the Pass1 variable is used incorrectly.
PicH	In geoAssembler, an internal variable which represents the height (in pixels) of the most recently defined bitmap.
PicW	In geoAssembler, an internal variable which represents the width (in bytes) of the most recently defined bitmap.
position independent code	Machine code which is designed to be loaded and executed anywhere in the 6502 memory space. It contains no absolute references (such as a **jmp** instruction) within the code area. Compare with *relocatable object code.*
program counter	See *PC.*
psect	Program section in geoAssembler; manages program code and initialized data.
pseudo-op	See *directive.*
ramsect	RAM section in geoAssembler; manages unitialized data space.
.rel	file name extender for relocatable object code output by geoAssembler and relocated by geoLinker.
relative address	An address which is specified in relation to another address; geoAssembler relocatable object code is stored in a relative format — references are relative to the psect base address; 6502 branch instructions use relative addressing — references are relative to the current instruction.

relocatable object code	The files output by geoAssembler are not assembled to run at a specific address — all the relative, relocatable addresses must be adjusted in the linker.
resident module	In a VLIR application, record zero which is loaded and run when the application is opened from the deskTop.
resolve	To evaluate an expression; to match up an external reference with a global label in another file.
scope (of labels)	The region within source code where a label may be referenced. The scope of a local label is the area between two successive global labels. The scope of a global label is always the entire current assembly file; its scope will be extended to files linked within the same module unless suppressed with the **.noglbl** assembler directive.
sequential application	A type of GEOS application where the program loads entirely into memory and does not support overlay modules. Compare with *VLIR application.*
source code	A geoWrite file for geoAssembler which contains 6502 assembly language.
swap module	See *overlay module.*
.sym	File name extender for a viewable symbol file (geoWrite compatible).
symbol	A label or an equate.
symbol table	A table which contains all the symbols for an application.
symbolic debugger	A debugger, like geoDebugger, which uses your applications symbols to display memory and disassembled machine code.

syntax	The format of a command or line of source code.
true	A logical truth-value: if an expression is true, it evaluates to an arithmetic one ($0001) and an arithmetic non-zero value ($0001-$ffff) is considered true. Compare with *false*.
truth value	The result of a logical expression — true or false.
two-pass assembler	An assembler, like geoAssembler, which makes two passes through the source code, accumulating labels, equates, and macros on the first pass and resolving references and generating object code on the second pass.
two's complement	The one's complement of a number plus one, where all one's become zeros and all zeros become ones and one is added to the result. In the *two's complement numbering system*, a negative number is the two's complement of its positive counterpart; taking the two's complement of a number is identical to subtracting it from zero. Compare with *one's complement*.
unitialized data	Data areas reserved in zsect and ramsect sections with the **.block** directive. No object code is generated, so the data space, although reserved, is not initialized with any values.
V	Overflow flag. Bit six in the 6502 status register. If set, the arithmetic operation generated an overflow.
VLIR application	A type of GEOS application where the resident module loads entirely into memory and the remainder of the code is swapped in and out as overlay modules.
wolf-fence method	A debugging technique where a bug is located by successively fencing it into smaller and smaller areas of code.
word	Two bytes combined to form one 16-bit value. Words are usually stored in low/high order.

X	One of the two 6502 index registers.
Y	One of the two 6502 index registers.
Z	Zero flag. Bit one in the 6502 status register. If set, the operation generated a zero.
zero page	Page zero in the 6502 memory space ($00-$ff); special because memory loads and stores to these locations are quicker than in the remainder of addressable memory.
zsect	a section in geoAssembler which manages zero page unitialized data space.

Index

Operators

geoAssembler Directives

geoLinker Directives

Super-debugger Commands

Mini-debugger Commands

Non-alphabetic

A

B

C

Q

R

S

T

U

V

W

Z