Two Assemblers For GEOS

A comparison of GeoCOPE and Geoprogrammer

by Francis G. Kostella

GeoCOPE is available for $20 (US) from:

Bill Sharp Computing
P.O. Box 7533
Waco, TX 76714

Geoprogrammer is available from:

Berkeley Softworks
2150 Shattuck Ave.
Berkeley, CA 94704

I'd like to present you with a brief overview of two assemblers that run under GEOS, and then throw a short argument at you as to why *you* should be writing programs for GEOS.

If you've written programs for GEOS with a non-GEOS assembler, you are probably familiar with the time-consuming hassle that the programmer must endure in order to convert his object file into a runnable GEOS file. Besides the object file needing to be manipulated in order to work with GEOS, one must constantly exit GEOS to make modifications and wait while the assembler creates the new file, usually without the speed that the TurboDOS adds, then re-boot the GEOS system.

What the GEOS programmer needs is a system that operates while in the GEOS environment and outputs GEOS-ready files. There are presently two packages (that I know of!) that do this: **geoCOPE** from Bill Sharp Computing and **Geoprogrammer** from Berkeley Softworks.

Looking into GeoCOPE

GeoCOPE is the less complex of the two and was the first GEOS-specific assembler released. The **COPE** system has two main programs, *Editor* and *copeASM* (the assembler).

The nicest part of the COPE system is the *Editor*. **COPE** uses its own unique structure for its source files, and each page of a source file can hold up to 8K of text. A feature that I found very useful, was the *Editor*'s ability to make the source files either GEOS SEQ or VLIR. Thus, my system equates file would be SEQ structure and would be **.included** in the main VLIR source file. A VLIR file can have 127 records, and at 8K per record, you can see that each of your source files can get very large if needed.

A few other features of the *Editor* that I liked and found very useful were the ability to set a Bookmark, so that you could return to the last line edited, and an Autosave feature that automatically updated changes when selected. The biggest advantage of the *Editor* is that it is fast! Unlike *geoWrite*, you can quickly scroll up or down through a document, and since it doesn't support fonts or font styles you're not left waiting while the system calculates 24-point bold outlines. The *Editor* also supports Text Scraps, has a Save & Replace function, and allows you to save single pages of a VLIR source file as single SEQ source files, and SEQ as pages of VLIR files. The **COPE** *Editor* also allows you access to Desk Accessories (the system comes with one: *HexCalc* a hex/dec/bin calculator) and works with an REU. The only thing missing from the *Editor* is the ability to use tabs to offset the opcodes and comments from the left margin.

One thing to be aware of when using **COPE**'s *Editor*, is that it stores each line of the source file as a null-terminated string. If you use the Text Scrap function to move text to or from *geoWrite*, you'll run into a few problems. Pasting a Scrap into a **COPE** file from a *geoWrite* document is simply taken care of by entering CRs where the line ends should be. But *geoWrite* will choke on **COPE** Scraps (the *Text Manager* has no trouble with them, though). I've managed to get around this problem by writing a simple filter program that strips out all the nulls except the final one.

COPE supports 21 different pseudo-ops, from the typical **.BYTE** and **.WORD** to some GEOS-specific ones like **.ICON** for defining header icons and **.SEGMENT** for mapping out any VLIR modules. **COPE** also allows you to define macros with the **.MAC** and **.MND** directives.

Labels can be up to 32 characters in length and are case sensitive. **COPE** also allows you to use local branch labels - up to 32 outstanding (unresolved) local labels are permitted. All the usual arithmetic and logical operators are supported (*, /, +, -, AND, EOR, OR).

The major advantage that the COPE system has is its simple and direct approach. Within an hour of reading the manual, I had an application up and running. The manual itself only details the features of the *Editor* and *copeASM*. The examples are few, but cover all the specifics. If you've used MADS, you'll adapt very quickly as the two are very similar.

The manual together with the sample files will have you writing VLIR applications in no time at all. By using the **.SEGMENT** directive, you simply indicate the end of one VLIR module and the start of another. The nice thing about this approach is that labels in all the different modules are global and the need for jump tables or duplicate label definitions is eliminated.

COPE files are easy to maintain and update. I've kept all my COPE source in VLIR structure: the first page holds the header block definition and an index to the entire file, the second page holds my constants and equates, the third page holds all of my macros, and the source code begins on page four. Writing even the largest of applications, I've never gone over 20 VLIR pages of source.

Once you have your source code together, you load the assembler and select the file. *CopeASM* is a two-pass assembler that will assemble to disk and print any errors to the screen. You may list the assembly on its second pass and can turn this listing on or off. *CopeASM* also allows you to pause the listing if needed. If the file assembles without any errors, you can exit to DeskTop and run the file. If you do have errors, you'll have to pause the assembler and scribble them down. This is the weakest part of the system - an option to output to an error file is sorely needed. Furthermore, attempting to pause the screen listing will sometimes scroll the errors off the screen, forcing you to reassemble just to see the errors.

If you're just getting your feet wet with GEOS, GeoCOPE is the perfect place to start...

Presently, **COPE** can only handle up to 5000 characters in its label table, so you'll have to keep those labels short and use plenty of local labels if you are assembling a large application. Another limit of *copeASM* is that it can only assemble 8K sections of code at a time. This isn't as big a problem as it may seem, though: you can simply divide the file up into VLIR modules and load them in as part of your initialization routine. My favorite approach here is to put all of my tables, graphics, and fonts into one or more VLIR modules and load them in right after drawing the intro screen.

Although I haven't done any rigorous comparisons of assembly times, I've got one good example: the original version of CIRCE was assembled with the MADS assembler, then converted to GEOS format. The amount of time taken between loading the MADS assembler and loading the assembled and converted file from the DeskTop was slightly under 35 minutes. After converting the source files to **COPE** format, the time between loading the **COPE** assembler and loading the assembled application was just about four minutes! Now I've got to admit that this was on an REU, but even without it, the assembly took about six or seven minutes. (Besides, MADS didn't support the REU.)

If you're just getting your feet wet with GEOS, COPE is the perfect place to start. The system easily handles smaller programs and is fairly fast and easy to maintain. But once your applications begin to grow in size, that 5,000 character symbol table begins to fill up fairly quickly. Once you get to this point, you should consider **Geoprogrammer.**

The first thing that strikes you when you open the **Geoprogrammer** package is the 450-page manual. If you have a number of GEOS programs, you are perhaps used to the sometimes simplistic documentation that presents you with the "this is a disk, put the disk in the drive..." level of information. I was very pleasantly surprised to open this manual and read through it without ever having my intelligence insulted. The first two dozen pages do contain the basic info for first time GEOS users, and there is a chapter devoted to an overview of the **Geoprogrammer** system, but the rest of the manual is filled with loads of useful information. Granted, the info is not all organized as well as it might be ("Now, *where* is that part about bitwise exclusive-or?!") but generally you'll be able to find what you need with a little persistence. In addition, the appendices of the manual contain detailed listings of all the system constants and variables, along with a hardcopy listing of the macro and sample source files included on disk.

The **Geoprogrammer** disk itself is a flippy that includes, besides the sample files and system symbols and macros, the three programs that make up the **Geoprogrammer** system: *GEOASSEMBLER*, *GEOLINKER* and *GEODEBUGGER*. No, there is no editor here. Unless you have a text editor that handles *geoWrite* files *[such as Q&D Edit, written by Kostella and Buckley and available from RUN - Ed.]*, you'll have to edit your source files with *geoWrite*! And **Geoprogrammer** is a two-drive system; you *can* use it with one drive, but the amount of disk swapping involved will quickly convince you that that second drive is worth the money. Better yet, an REU is not only a fast second drive that makes using *geoWrite* tolerable, but it will allow you to use *GEODEBUGGER* to its fullest.

That being said, let me give a brief overview of the assembly process. Once you've edited all of your source, you assemble each of the source files into *.rel* relocatable object files. Then you load a linker file (also a *geoWrite* document) into *GEOLINKER* to link together your *.rel* files into a runnable GEOS file on disk. Now you can load *GEODEBUGGER* to test and debug your program. Sounds simple, eh? Well, there's a lot more going on here than would appear. **Geoprogrammer** gives you access to some powerful features and abilities that you may find that you won't want to do without once you've gotten used to them.

First off, *GEOASSEMBLER* is a two-pass assembler that supports a number of useful features: conditional assembly, macros, local labels, the ability to parse complex algebraic expressions, and the ability to pass symbols to the linker or debugger. *GEOASSEMBLER* will also output an error file to disk (in *geoWrite* format, of course!), if needed, for each file assembled.

When *GEOASSEMBLER* starts assembling a file, it uses three counters to keep track of the code: **.zsect** for zero page ram, **.psect** for program code, and **.ramsect** for uninitialized data. If you're lazy like I am, when you need a new variable, you just add it somewhere in the current section of code instead of adding it to a separate section of code for variables. By using the **.psect** and **.ramsect** directives, you can add variables just about anywhere like this:

```
.ramsect
MyVariable:    .block 1
.psect\b
```

When the assembler encounters this construction, it will give the label *MyVariable* the address of the current address of **.ramsect** (which can be set by the **.ramsect** directive or in the linker file). The **.ramsect** section defaults to the RAM following the last byte of code, thus we don't end up assembling uninitialized variables and add to the length of the program. Perhaps not a big deal, but when you have a few hundred bytes of variables it becomes noticeable.

Another useful feature of the assembler is the 16-bit expression evaluator (*GEOLINKER* also uses the same evaluator). Besides the usual arithmetic, the evaluator handles a number of logical operators: the manual lists thirty of them. I usually keep away from creating expressions too complex to be understood at one glance, but *GEOASSEMBLER* will let you create some truly bizarre and outlandish expressions if you so desire! But the real power I find here is that you can easily create data tables with a few easily changed constants at the root of some complex expressions. Perhaps this doesn't seem that unusual, but I've been able to create expressions that all the other assemblers I own have choked on, and I don't miss having to do the math by hand.

You run *GEOASSEMBLER* from DeskTop and select the file to assemble from the typical 15-file dialog box. Once you've selected the file to assemble, you are given a choice of drive for the output file, then the file is assembled. The output file is the same name as the source file but with a *.rel* appended. When this is done, you can quit to DeskTop, assemble another file, or open the error file (i.e. enter *geoWrite*) if one was generated. A friend of mine who beta-tested the version 2.0 package tells me that *GEOASSEMBLER* V2.0 will allow you to go directly to the linker, and that it is not limited to selecting only the first 15 source files on disk.

Once you've assembled all the *.rel* files in your program, it's time to use *GEOLINKER*. *GEOLINKER* does more than just connect separate object files together. First of all, *GEOLINKER* uses a link file to determine the structure of the output program, be it GEOS SEQ, VLIR, or CBM, which is a 'regular' object file. Secondly, the linker will add the header to the file. The linker will also cross-resolve all label references between the different *.rel* files; if a label is defined in two different files, but is not referenced in a third file *GEOLINKER* will not flag an error.

One thing I would like to mention here is that although the assembler and linker accept symbol names up to 20 characters in length, only the first eight are significant. This is not really a problem, but you should remember that there are a few hundred system symbols in the *geosSym* file usually *.include*d during assembly.

I once wrote a routine named *DeleteRegion* that was only called from a routine in another source file. *DeleteRegion* never seemed to be called, but the disk would go active for a second when it should have been called. The debugger only shows the eight significant characters of the label when you list the code, so I couldn't imagine what was happening. But the debugger also lets you view the label name by its hex address, and upon examination, this label appeared somewhere in the Kernal jump table. The routine that was being called was the Kernal routine *DeleteRecord*! Luckily I didn't have any VLIR files open....

GEOLINKER will also allow you to output a separate symbol table (again, a *geoWrite* file) to the drive of your choice. Like *GEOASSEMBLER*, if there are any errors, you have the option of directly opening the error file after linking. One thing to note here is that when there are more than 99 errors, the system will sometimes have a fatal crash.

When *GEOLINKER* creates the file on disk, it also writes a special *.dbg* file to disk for use by the debugger - more on this later.

The manual does not include specifications for either *GEOASSEMBLER* or *GEOLINKER*, otherwise I'd be happy to include a list here. If you do run into problems assembling and linking very large files, you can always use the **.noglbl** and **.noeqin** directives to cut down the number of symbols passed to the linker. One way to quickly cut down on the number of symbols is to use **.noglbl** and **.noeqin** before **.include** *geoSym* (the system labels and equates) and **.glbl** and **.eqin** after. Of course you don't get anything for free, and these symbols don't get passed to the debugger!

A sneaky trick I use when doing this, to get the system symbols to the debugger when writing a large VLIR application, is to assemble the *geosRoutines* and *geosMemoryMap* and link them into one of my modules that isn't using very many symbols. Each module has its own set of symbols and is selected in the debugger by the **setmod** command. If you're debugging a stretch of code that uses a lot of system calls, just reset the module priority to the module with the system symbols.

Alternately, I find that just defining the pseudo-registers as global equates helps a great deal when debugging.

Another potential problem I've noticed that is not documented, is that when linking files from two drives, the linker seems to search for the file on the current drive first, then checks the other drive. The problem here is that if you have two different versions of one particular *.rel* file, let's call it *beta.rel*, and this file is supposed to be linked after a file called *alpha.rel* that is not on the current disk, when the linker switches drives to find *alpha.rel*, it does not switch back to the original drive, potentially linking the wrong *beta.rel*. I don't have any empirical evidence for this, but if your modifications don't seem to be appearing in your new version of a program, try moving all of the *.rel* files in one VLIR module to the same disk.

Once you've assembled a GEOS program, it's time to load *GEODEBUGGER* - here's where the fun begins! *GEODEBUGGER* is a sort of 'shell' that uses NMIs to take control of the machine and allows you to debug the program (or examine the Kernal) in an almost interactive environment. Load your program from the debugger. Need to tweak some values or check why that branch isn't being taken? Just bang on the RESTORE key and you're in the *GEODEBUGGER* again. *GEODEBUGGER* is the best thing about this package; you can set breakpoints, alter stack or register values, and access the disk drives almost like a sector editor.

Geoprogrammer along with an REU will dramatically increase your output and capabilities...

There are actually two versions of GEODEBUGGER. When you load the program, it first checks for a REU. If there is one connected, the full debugger is loaded into the REU, otherwise the mini-debugger is loaded into RAM from $3E00 to $5FFF. Needless to say, the REU super-debugger is the preferred option.

What makes the debugger truly useful is the ability for it to use the *.dbg* files generated by the linker. These files contain a list of symbols and their addresses. This way, while in the debugger, you can list and modify a section of code using labels from your source files. Of course, your changes are not saved, but this allows you to try out different things without constantly reassembling the program.

GEODEBUGGER is basically a machine language monitor with plenty of features. One of the most powerful of these in the super-debugger is the ability to define macros. Most of the commands in the debugger are actually system macros composed of a number of macro primitives. *GEODEBUGGER* allows you to define up to 1,000 bytes of user macros. These user macros can be made up of the macro primitives or system macros. A macro file with the same name as your application will be automatically loaded along with your application. Optionally, you can define a default set of macros and an autoexec macro to run when the debugger is loaded. For example, the linker always passes a few of the system variables to the debugger, but I have no use for them, so my autoexec macro removes these from the symbol table, like this:

```
.macro autoexec     ; name
   clrsym Pass1[cr] ; eliminate symbol
   clrsym picW[cr]
   clrsym picH[cr]
.endm
```

There are dozens of other commands, but there are problems with some of the memory commands, most notably FILL, which doesn't work at all. My beta-test friend tells me that this bug has been eliminated in the 2.0 version. The only other thing that one could desire would be a more detailed description of the macro primitives in the manual.

Overall, the **Geoprogrammer** package is nicely put together, and along with an REU will dramatically increase your output and capabilities. Most especially, the debugger will teach you about programming for GEOS by allowing you to examine any GEOS program and the GEOS Kernal in detail.

If you're new to assembly language, I suggest that you give writing GEOS programs a try. A common problem for beginners is the need to develop a set of routines to perform common functions; i.e., printing text and graphics to the screen, moving large chunks of memory around, disk access, and string input to name just a few. When you're just starting out, you basically begin with nothing, and until you've accumulated enough experience to write code to perform some of the above functions, your ability to write useful programs is hindered. When you code for GEOS, all of these basic functions are always available. You can concentrate on writing the 'heart' of the program without getting bogged down in minor details. By getting programs up and running quickly, the beginner will (hopefully) form positive associations with assembly language, instead of thinking of it as some arcane art which is painfully learned!

If you're interested in writing GEOS programs, you must get an assembler package that runs in GEOS. If you're not sure how far you want to go I suggest you get the **geoCOPE** assembler from Bill Sharp Computing. The price is good and the system direct and uncomplicated. If you later decide that you need more power and you have a two-drive system (or REU!) and can afford the price, go for **Geoprogrammer.** If you're an experienced programmer, I suggest that you go straight to **Geoprogrammer** or get them both.

Berkeley Softworks should be commended for releasing such a nice package, especially considering the relatively small market for products of this nature. Unfortunately, they don't seem as if they're going to release version 2.0 any time soon. One only hopes that they would at least consider doing a mail-in upgrade for present users.