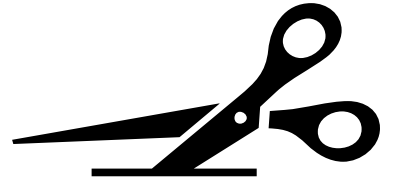


Tools



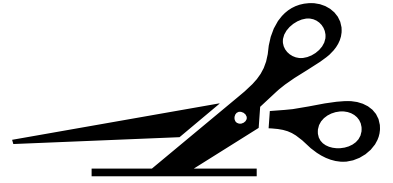
Version 2.0



GEOS Software Development Kit Library
Version 2.0



Tools



Initial Edition, Unrevised and Unexpanded

Geoworks, Inc.
Alameda, CA



Geoworks provides this publication "as is" without warranty of any kind, either express or implied, including but not limited to the implied warranties of merchantability or fitness for a particular purpose. Geoworks may revise this publication from time to time without notice. Geoworks does not promise support of this documentation. Some states or jurisdictions do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

Copyright © 1994 by Geoworks, Incorporated.
All rights reserved. Published 1994
Printed in the United States of America

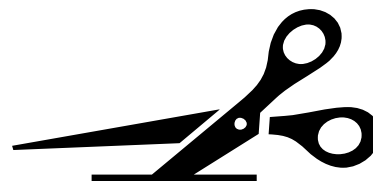
Geoworks®, Geoworks Ensemble®, Ensemble®, GEOS®, PC/GEOS®, GeoDraw®, GeoManager®, GeoPlanner®, GeoFile®, GeoDex® and GeoComm® are registered trademarks of Geoworks in the United States and Other countries.

Geoworks® Pro, PEN./GEOS, Quick Start, GeoWrite, GeoBanner, GeoCrypt, GeoCalc, GeoDOS, Geoworks® Writer, Geoworks® Desktop, Geoworks® Designer, Geoworks® Font Library, Geoworks® Art Library, Geoworks® Escape, Lights Out, and Simply Better Software are trademarks of Geoworks in the United States and other countries.

Trademarks and service marks not listed here are the property of companies other than Geoworks. Every effort has been made to treat trademarks and service marks in accordance with the United States Trademark Association's guidelines. Any omissions are unintentional and should not be regarded as affecting the validity of any trademark or service mark.



Contents



1	Welcome	11
2	System Configuration	13
2.1	Development Directories and Files	TConfig : 15
2.2	Target Directories and Files	TConfig : 19
2.3	Environment Variables	TConfig : 21
2.4	System Files	TConfig : 22
2.5	Sample C Applications	TConfig : 23
2.6	Sample Esp Programs	TConfig : 29
3	Swat Introduction	31
3.1	DOS Command Line Options.....	TSwatCm : 35
3.2	Notation	TSwatCm : 36
3.3	Address Expressions.....	TSwatCm : 37
3.4	On-line Help	TSwatCm : 39
3.5	Essential Commands	TSwatCm : 42
3.5.1	Cycle of Development	TSwatCm : 42
3.5.2	Attaching and Detaching.....	TSwatCm : 43
3.5.3	Breakpoints and Code Stepping	TSwatCm : 47
3.5.4	Examining and Modifying Memory	TSwatCm : 54
3.5.5	Other Important Commands	TSwatCm : 65
3.6	Additional Features.....	TSwatCm : 69
4	Swat Reference	73
4.1	Notation	TSwtA-I : 75
4.2	Swat Reference.....	TSwtA-I : 76



5	Tool Command Language	263
5.1	Using This Chapter	TTCL : 265
5.2	Copyright Information	TTCL : 266
5.3	Background and Description.....	TTCL : 266
5.4	Syntax and Structure	TTCL : 267
5.4.1	Basic Command Syntax.....	TTCL : 268
5.4.2	Expressions	TTCL : 273
5.4.3	Lists.....	TTCL : 275
5.4.4	Command Results.....	TTCL : 275
5.4.5	Procedures	TTCL : 276
5.4.6	Variables	TTCL : 277
5.5	Commands	TTCL : 277
5.5.1	Notation.....	TTCL : 277
5.5.2	Built-in Commands	TTCL : 278
5.6	Coding	TTCL : 302
5.6.1	Swat Data Structure Commands	TTCL : 303
5.6.2	Examples.....	TTCL : 328
5.7	Using a New Command.....	TTCL : 330
5.7.1	Compilation.....	TTCL : 331
5.7.2	Autoloading	TTCL : 331
5.7.3	Explicit Loading	TTCL : 331
6	Debug Utility	333
6.1	Changing Platforms.....	TDebug : 335
6.2	Switching Kernels	TDebug : 336
7	Icon Editor	337
7.1	Creating Icons.....	TIconEd : 339



7.2	Importing Icons	TIconEd : 340
7.3	Editing Icons.....	TIconEd : 340
7.4	Writing Source Code	TIconEd : 341
7.5	Icon Databases	TIconEd : 342
7.6	Exporting to Database	TIconEd : 343
8	Resource Editor	345
8.1	Glossary	TResEd : 347
8.2	Getting Started	TResEd : 350
8.3	What Needs to be Translated?	TResEd : 350
8.4	Translating	TResEd : 350
8.4.1	Choosing a new translation file	TResEd : 351
8.4.2	Main translation screen.....	TResEd : 351
8.4.3	Translating a Text String	TResEd : 352
8.4.4	Moving between chunks	TResEd : 353
8.4.5	Moving between resources.....	TResEd : 353
8.5	Resource Editor Menus	TResEd : 354
8.5.1	File Menu	TResEd : 354
8.5.2	Edit Menu	TResEd : 355
8.5.3	Project Menu.....	TResEd : 355
8.5.4	Filter Menu	TResEd : 356
8.5.5	Utilities Menu	TResEd : 356
8.5.6	Window Menu	TResEd : 357
8.6	Creating an Executable	TResEd : 357
8.7	Updating an Executable.....	TResEd : 358
8.8	Testing Your New Executables	TResEd : 359



9	The INI File	361
9.1	How to Use the INI File	TIni : 363
9.2	Categories in the INI File	TIni : 364
9.2.1	[cards]	TIni : 370
9.2.2	[configure]	TIni : 371
9.2.3	[diskswap]	TIni : 372
9.2.4	[envelope]	TIni : 373
9.2.5	[envel<num>]	TIni : 374
9.2.6	[expressMenuControl]	TIni : 374
9.2.7	[fileManager]	TIni : 376
9.2.8	[input]	TIni : 378
9.2.9	[keyboard]	TIni : 382
9.2.10	[label]	TIni : 384
9.2.11	[label<num>]	TIni : 385
9.2.12	[link]	TIni : 386
9.2.13	[localization]	TIni : 387
9.2.14	[math]	TIni : 387
9.2.15	[modem]	TIni : 387
9.2.16	[<modem name>]	TIni : 388
9.2.17	[mouse]	TIni : 391
9.2.18	[net library]	TIni : 392
9.2.19	[paper]	TIni : 392
9.2.20	[paper<num>]	TIni : 393
9.2.21	[parallel]	TIni : 394
9.2.22	[paths]	TIni : 394
9.2.23	[printer]	TIni : 396
9.2.24	[<printer device name>]	TIni : 397
9.2.25	[screen 0]	TIni : 399
9.2.26	[serial]	TIni : 401



9.2.27	[sound].....	TIni : 402
9.2.28	[spool]	TIni : 402
9.2.29	[system].....	TIni : 403
9.2.30	[text].....	TIni : 409
9.2.31	[ui].....	TIni : 413
9.2.32	[<specific ui name>]	TIni : 421
9.2.33	[ui features]	TIni : 421
9.2.34	[welcome].....	TIni : 428

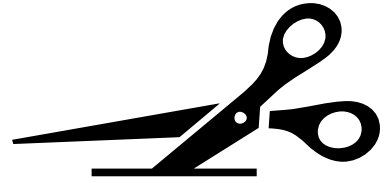
10	Using Tools	431
10.1	Tools Summary	TTools : 433
10.2	Typical Development Session.....	TTools : 434
10.3	File Types.....	TTools : 435
10.4	Esp.....	TTools : 439
10.5	Glue.....	TTools : 440
10.6	Goc	TTools : 443
10.7	Grev	TTools : 444
10.8	mkmf	TTools : 447
10.9	pccom.....	TTools : 448
10.9.1	PCCOM Background.....	TTools : 448
10.9.2	Running PCCOM on the Target	TTools : 449
10.9.3	File Transfer Protocol of PCCOM	TTools : 455
10.10	pcget	TTools : 461
10.11	pcs	TTools : 462
10.12	pcsend.....	TTools : 465
10.13	pmake	TTools : 465
10.13.1	Copyright Notice and Acknowledgment	TTools : 466



10.13.2	How to Customize pmake.....	TTools : 467
10.13.3	Command Line Arguments	TTools : 469
10.13.4	Contents of a Makefile	TTools : 471
10.13.5	Advanced pmake Techniques	TTools : 493
10.13.6	The Way Things Work.....	TTools : 497
10.14	Swat Stub	TTools : 498
	Index	ToolsIX : 499



Welcome



1

This manual provides full information about the GEOS development tools. It also details your system configuration (directory tree, files, etc.) after you've finished installing the tools. This manual is intended as a reference guide. You should begin learning about GEOS programming with the tutorial.

The five main sections of this manual are

System Configuration

A listing of the directories and files that are on your machines after installation. This section also gives an overview of all the sample applications, where they're located, and their purposes.

Swat and TCL

Documentation for Swat, along with in-depth reference for TCL, a language by which you can write Swat commands.

GEOS Utilities

Documentation for some GEOS developer utilities which run in the environment.

The GEOS.INI File

A listing of all the categories, keys, and options allowed in the GEOS.INI and GEOSEC.INI files.

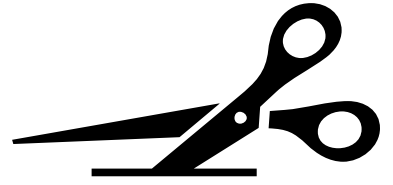
Tools

A reference section detailing the command-line usage and options of all the tools.





System Configuration



2.1	Development Directories and Files	15
2.2	Target Directories and Files	19
2.3	Environment Variables	21
2.4	System Files	22
2.5	Sample C Applications	23
2.6	Sample Esp Programs	29

2





This chapter details how your software layout should look after installation. This information is provided both so that you can confirm your setup and so that you can make appropriate modifications later.

2.1 Development Directories and Files

2.1

The installation program will install a large number of tool files, debugging files, and sample applications on to the development machine. These are arranged in the directory of your choice (with name specified in the `ROOT_DIR` environment variable). Most developers will use `\PCGEOS` as the root directory of their development tree, and this is the name assumed throughout this section.

`\PCGEOS` This directory is the root of the development tree. It does not contain any files, but all the development files are in subdirectories of this root. The first two levels of the PCGEOS directory tree are shown in Figure 2-1.

`\PCGEOS\LIBRARY`

This directory houses a number of subdirectories, each of which contains the executable and symbolic information files for a GEOS library. Some of the library directories contain subdirectories. For example, the `MATH` directory contains a subdirectory `COMPILER` that contains files for different C compilers. For the most part, you will not need to look in these subdirectories unless you need to re-download the geodes.

`\PCGEOS\DRIVER`

This directory contains a number of subdirectories, each of which corresponds to a single type of driver (e.g. file system driver, printer driver, mouse driver, etc.). Each of these subdirectories contains the executable and symbolic files for the individual GEOS drivers (e.g. the `\PCGEOS\DRIVER\MOUSE\MSBUS` directory contains the files for the Microsoft bus-type mouse drivers). For the most part, you will not need to look in these subdirectories unless you need to re-download the geodes.



System Configuration

16

2.1

```
/PCGEOS

/LIBRARY
/KERNEL      /PREF      /FLATFILE    /TEXT      /CARDS      /MOTIF
/GROBJ       /CONVERT   /CHART       /SPELL     /IMPEX      /MATH
/USER        /PARSE     /PEN         /SPLINE    /RULER      /BITMAP
/COLOR       /SOUND     /HWR         /CONFIG    /SSMETA
/COMPRESS    /GAME      /FMTOOLS     /ANSIC     /CELL
/SWAP        /STYLES    /NET         /SPOOL     /SHELL

/DRIVER
/PRINTER     /MOUSE     /KEYBOARD    /IFS       /VIDEO      /FONT
/SOUND       /POWER     /SWAP        /SDK       /TASK       /STREAM
/NET         /DMA       /PCMCIA      /FAX

/INCLUDE
/LDF         /OBJECTS   /ANSI        /INTERNAL   /SDK_C

/LOADER

/APPL
/FILEMGRS    /ICON      /PREFEREN    /SDK_C     /STARTUP    /TOOLS      /CALC
/CONSUM      /USINFO    /ALARM       /DEMO      /GEOFILE    /WINFO      /NOTE
/SCRAPBK     /FCALC     /GEOWRITE    /DUMP      /LANGX      /SDK_ASM    /DICT

/BIN

/TCL

/EXTRA
```

Figure 2-1 Host Directory Structure

The top development directory is \PCGEOS. The seven subdirectories contain files and directories necessary for development.

\PCGEOS\INCLUDE

This directory contains definition and header files which your programs will include, as well as a few subdirectories. Among these are kernel header files, library header files, and system makefiles. You will probably spend a lot of time looking in these files as well as in the documentation.

The .GOH files are Goc header files. The .MK files are Makefiles, used by the PMake utility. The .DEF files are definition files included by Esp programs. Glue uses the .PLT files to represent software platforms.



The subdirectories of the \PCGEOS\INCLUDE directory are

LDF	This contains the library definition files for all GEOS libraries. When you are ready to install a library, you will create a library definition (.ldf) file which will reside here. You will not access these files directly.
OBJECTS	This contains files associated with GEOS object libraries and GEOS classes. You will probably look here often for calling conventions and features of certain classes.
ANSI	This contains files used for standard ANSI C functions. You may look through these files occasionally.
INTERNAL	This contains files which are normally GEOS-internal. You will only look through these files if you are doing driver development.
SDK_C	This directory contains include files meant as sample code.

2.1

\PCGEOS\LOADER

This directory contains the executable and symbol files for the GEOS loader. This is the program which loads the GEOS kernel. You will not need to look in this directory unless you need to re-download the loader. Subdirectories contain the Zoomer and PT9000 loaders.

\PCGEOS\APPL

This directory contains a number of subdirectories, each of which contains either a single application or a set of subdirectories for a category of application. The sample applications (in the SDK_C subdirectory) include source code but not executable or symbol files. The other applications include only the executable and symbol files.

The subdirectories of the \PCGEOS\APPL directory are

DUMP	This directory contains the GEOS screen dumper.
FILEMGRS	This directory contains the various file manager applications.
GEOMANAG	This directory contains the GeoManager application which is the GEOS file manager normally used on desktop machines.
ZMANAGER	This directory contains the Zoomer file manager, suitable for palmtop machines.

System Configuration

18

2.1

ICON	This directory contains the executable and symbol files for the Icon editor tool. The icon editor is installed on the target machine; you must run it on the target, like other applications. You will not likely need to access this directory.
PREFEREN	This directory contains the standard applications used for setting user preferences. There are four subdirectories: SETUP, PREFMGR, ZPREFMGR, and ZSETUP. The first two contain the setup and Preferences applications for desktop machines. The second two contain the setup and Preferences applications for the Zoomer. You should not need to access these directories.
SDK_C	This directory contains the source code for all the GEOS sample applications. You will probably spend a lot of your time browsing, editing, and studying the sample applications. This directory contains a number of subdirectories; each contains either a single application or numerous applications, each in a further subdirectory. The applications included are outlined in "Sample C Applications" on page 23.
STARTUP	This directory contains the applications and data needed by the system at startup. Specifically, it contains a subdirectory with the Welcome application's executable and symbol files. You will not likely need to access this directory.

\PCGEOS\BIN

This directory contains the DOS executable tools you will use to construct GEOS applications. (This directory should be appended to your PATH environment variable.) You should not need to access this directory directly.

\PCGEOS\TCL

This directory contains Tcl (Tool Command Language) scripts. The Swat debugger will use these scripts, which provide functions useful for working with GEOS data structures. You should only need to access this directory if you write your own Tcl code for Swat or if you want to look at or edit the provided Tcl files.

The EXTRA directory in this directory contains additional Tcl scripts that are both undocumented and unsupported. They are provided because you might find them useful even though they are unsupported.



2.2 Target Directories and Files

The target machine install will set up a standard GEOS environment. It will set up the proper directory structure; note that this structure is important—the **pcs** tool, which automatically downloads geodes to their proper directory assumes that those proper directories exist. Also note that the install program will set up two trees: one with error-checking code, and one with normal code. When the instructions tell you to do something from the top-level target directory, then you should be in the top-level EC directory if using error checking code and the top-level normal directory when testing regular code.

2.2

Note that the top-level directory of your target install (both top level directories, in fact) will contain a program file SWAT.EXE. This is a program known as the Swat Stub—it will communicate with the actual Swat program, which will run on the development machine.

Throughout the directory tree will be files named @DIRNAME.000. These files contain GEOS-internal information about the directory name and links to other directories and files. These files are created and maintained dynamically by GEOS.

The standard GEOS setup includes the following directories:

- DOCUMENT Normally used to hold documents. This will most likely be empty after installation.
- PRIVDATA Normally used to hold data which the user should not work with directly. This directory contains the following subdirectories:
 - BACKUP Contains backup files created by the document control object (a standard feature of the document control).
 - HWR Contains handwriting recognition data.
 - LOGO Contains logo picture information.
 - PREF Contains information stored by the Preferences manager.
 - SPOOL Contains files maintained by the spool library (this normally consists of files containing data which will be sent to the printer).

System Configuration

20

2.2

STATE	Contains the “state” files that running applications leave behind when GEOS shuts down.
WASTE	Contents of the wastebasket. This directory will be created automatically when you drop a file or directory on the waste basket icon in GeoManager.
SYSTEM	Normally contains kernel and library geodes. If you create libraries to support your applications, you will most likely need to install them to this directory. This directory contains the library executables as well as several subdirectories for driver executables. The subdirectories are listed below (files are not listed here):
FILEMGR	Contains file manager tools. If you create file manager tools, you should install them here.
FONT	Contains font drivers. Note: Does not contain font definition files; the fonts themselves go in \USERDATA\FONT.
FS	Contains file system drivers.
IMPEX	Contains import and export translation libraries.
KBD	Contains keyboard drivers.
MOUSE	Contains mouse drivers and pen drivers.
PREF	Contains Preferences modules. If you create a Preferences module, you should install it here.
PRINTER	Contains printer drivers.
SAVERS	Contains individual screen savers.
SOUND	Contains sound drivers.
SWAP	Contains memory swapping drivers.
SYSAPPL	Contains special system applications such as Welcome and Graphical Setup.
TASK	Contains task-switch drivers.
VIDEO	Contains video drivers.
USERDATA	Top-level directory normally used to hold data which the user will be allowed to change or add to. This directory contains the following subdirectories:



DECK	Contains all playing card deck data files.	
FONT	Contains all font data files, regardless of font format.	
HELP	Contains all help files.	
WORLD	Normally used to hold applications. The World directory has several subdirectories for different types of applications. These subdirectories are listed below:	
ASM	Contains compiled assembly applications downloaded from the host machine's \PCGEOS\APPL\SDK_ASM directory. This will be empty on installation.	2.3
C	Contains compiled C applications downloaded from the host machine's \PCGEOS\APPL\SDK_C directory. After you finish the installation procedures and first chapter of the tutorial, this should contain the HELLO.GEO application.	
Desk Accessories	Contains desk accessory applications such as the Calculator. Applications installed in this directory will act as "desk accessories," which will always be on top of other application windows.	
Productivity	Contains productivity applications such as GeoWrite and GeoDraw.	
Utilities	Contains utility applications that are not desk accessories. GeoManager, the icon editor, Preferences, and the screen dumper are all examples of utility applications; these should all be installed on your system.	

2.3 Environment Variables

To make development easier, the environment variables on your development machine should have the following additions.

PATH	The development kit's \PCGEOS\BIN directory should be appended to your PATH. Your C compiler directory (and its \BIN directory, if appropriate) should also be in your PATH.
------	--



System Configuration

22

ROOT_DIR This variable should contain the directory in which you installed your kit (normally C:\PCGEOS).

PTTY This variable will control communication between the development and target PC. It should be of the format

`PTTY=comPort,baudRate[,inter]`

where *comPort* is the number of the development machine's serial port you are using, *baudRate* is a baud rate (e.g. 38400 or 19200), and *inter* is the number representing the interrupt level at which your serial port is operating. If this is the standard interrupt level for that serial port, this value need not be specified. These fields are separated by commas only; there should be no spaces. A typical PTTY setup is

`PTTY = 2,38400`

Your target machine needs only one extra environment variable—a PTTY variable to control the target machine's side of communications with the development machine. It is set up in the same way as the development machine. However, you should use the number of the COM port by which the target machine is connected.

2.4 System Files

The \AUTOEXEC.BAT and \CONFIG.SYS files on the development machine should have certain changes made to them for the tools to function properly. These changes are normally invoked by the installation program, but they are listed below for reference.

The AUTOEXEC.BAT file should define the PATH, ROOT_DIR, and PTTY environment variables on the host machine. These three variables are described in the previous section. On the target machine, the AUTOEXEC.BAT file defines only the PATH and PTTY variables for the GEOS SDK.

The CONFIG.SYS file on the host machine defines the number of files and buffers DOS can have open at one time. For best results, the FILES should be set to something larger than 80, and BUFFERS should be set to something larger than 30. On the target machine, set them to similar numbers. The



exact numbers best for your system may be different. See your DOS manual for more information on these items.

2.5 Sample C Applications

In your development machine's \PCGEOS\APPL\SDK_C directory, there should be a number of sample applications. The documentation will refer to these applications to illustrate certain points, but you may find a brief description of each useful if you wish to browse.

2.5

- APPICON Illustrates an application moniker. See the header file in the Art subdirectory to see a typical application icon.
- BENOIT Acts as an example of a large-scale application. Has several source files, and some of the source code is in assembly.
- CLIPSAMP Demonstrates use of the clipboard. This illustrates what an application can do to define its own format for cutting and pasting and working with quick transfer.
- CUSTGEOM Shows some of the special positioning directives that you can use to override and guide the Geometry Manager. This application illustrates several of the "hints" used with GenInteractions and other generic objects to force certain arrangements of UI gadgetry.
- DBSAMP Illustrates the use of the database (DB) library. The application displays various pieces of data which are stored in data structures provided by the DB library.
- DOCUMENT This directory contains several subdirectories which deal with the document control objects in one form or another. The most basic of these programs is DocView, and this application is probably the best starting place for learning about document control objects.
 - DEFDOC Demonstrates the specification of a "default document" that should be opened or created when the application is invoked without a document to open.
 - DOCUI Shows that you need not use a GenView to display the contents of a document; this application uses other types of objects. As an important side effect, it also demonstrates how UI objects

System Configuration

24

		can send messages to the current document via the “model hierarchy.”
	DOCVIEW	Shows the most common usage of the document control objects; this application shows the display of the document through a GenView, with the GenDocument object acting as the VisContent part of the view/content pair.
2.5	DOSFILE	Shows how to use a DOS file, rather than a GEOS VM file for the document. The biggest difference when working with DOS files is that the document control cannot automatically provide save or revert functionality for you.
	MULTVIEW	Shows the display of the document through a GenView, with the GenDocument object acting as the VisContent part of the view/content pair. However, this application also includes display control objects which allow the simultaneous display of multiple documents.
	PROCUI	Shows how to use your process thread rather than a subclass of GenDocumentClass to display your document. The process thread, here, uses UI objects to display the contents of the document, rather than drawing to a GenView.
	PROCVIEW	Shows how to use your process thread rather than a subclass of GenDocumentClass to display your document. In this application, the process thread draws things through a GenView to display the contents of the document.
	SHAREDDB	Shows how to cope with a VM file that has been marked for multi-user access.
	VIEWER	Shows how you might write something that doesn't edit files but only displays them.
	FOCUS	Demonstrates the use of the focus hierarchy. The system maintains the notion of a focus to keep track of which object should receive keyboard events.
	FSELSAMP	Demonstrates the use of a GenFileSelector independent of a GenDocumentControl (normally the document control uses the file selector to allow the user to navigate the file system—this application has a free-standing file selector). Includes several extra file-related gadgets.



FSFILTER	Demonstrates the use of a file selector UI object in filtering its display of files and directories (beyond what is provided by the various file selector filtering attributes).	
GENATTRS	Demonstrates what can be done with the GenStates and GenAttrs fields of a generic object. This application illustrates the effects of changing the GenStates and three GenAttrs.	
GENDLIST	Demonstrates the use and management of a GenDynamicList object. A dynamic list allows the application to provide the user with a list of items of any size without the overhead of creating an object for each item.	2.5
GENDISP	Demonstrates the capabilities of the GenDisplay. The GenDisplay object is the top-most visible long-term user interface object in the system. Its subclass, GenPrimary, is the main window the user sees for an application.	
GENINTER	Demonstrates the use of a GenInteraction. GenInteractions are used to group other generic gadgets. Here you can see some dialog boxes and menus.	
GENITEMG	Demonstrates the various ways you can use the GenItemGroup and GenBooleanGroup objects to make lists from which the user can select items. An object of related usefulness, the GenDynamicList, is demonstrated in the GenDList application.	
GENTREE	Demonstrates manipulation of an application's generic tree. Includes adding and removing gadgetry to and from the tree.	
GLYPH	Shows the usage of a GenGlyph UI object. A GenGlyph is a simple UI object used most often only to display a visual moniker.	
GROBJ	This directory contains one subdirectory, DUPGROBJ, which in turn holds the dupgrobj sample application. This application shows the use of the Graphic Object (grobj) library together with document control objects.	
GSTEST	Demonstrates some of the GString macros used in the construction of graphics strings in visual monikers.	
GSTEST3	Shows a plethora of bitmap-based GString monikers.	
HELLO	Implements a Hello World style application in which colored text is drawn at an angle inside a scrolling window.	
HELLO2	Implements an expanded hello program with a text entry dialog that allows the user to change the text displayed.	

System Configuration

26

2.5

HELLO3	Implements an expanded hello program with triggers to change the text color.
HELP	This directory contains several sample applications showing how to use the help library.
HELPSAMP	This sample application shows some simple usage of on-line help in a C application. Note that along with the executable, you will have to download the helpsamp.000 , circles.000 , and squares.000 help files. These files will go in the target's help directory.
HELPtrig	This sample application shows how to introduce Help triggers in different kinds of dialog boxes. Note that along with the executable, you will have to download the helptrig.000 help file. This file will go in the target's help directory.
HELPVIEW	This application views a help file. It shows how you can use the help system as a HyperText viewer. Note that along with the executable, you will have to download the helpview GeoWrite file. You should use the help editor to generate the corresponding help file before running the program.
IACP	This directory contains a few applications which show how to use the Inter-Application Communication Protocol.
CLIENT1	This program works together with SERVER, providing the client side of a client/server pair of programs.
SERVER	This program works together with CLIENT1, providing the server side of a client/server pair of programs.
PALMSHUT	Example of a special purpose IACP application. This one will shut down any running copy of Palm's address book application for the Zoomer.
INKTEST	This application demonstrates simple use of an ink object.
LEVELS	Shows the use of the UI levels mechanism (which allows simpler UI for less computer-savvy users).
MONIKER	Demonstrates the various types of monikers a generic object can have and some of the various ways one can change the moniker an object displays. Also provides an example of a statically defined graphics string, and how to dynamically create a memory based graphics string.



MULTPRIM	Illustrates an application with more than one primary, as well as some ways an application may manipulate them. The application can change its own name and icons for each primary.	
PCCOM	These applications illustrate use of the pccom library.	
PCCOM1	Shows a fairly standard use of the pccom library, with the ability to exchange files over the serial line.	
PCCOM2	This application loads the pccom library dynamically. It illustrates how to load a library dynamically and then use that library's entry points.	2.5
PSCTEXT	Demonstrates the use of a PointSizeControl with a text object. Note that this same approach can be used to cause almost any text-related controller object to communicate with a text object.	
SERIAL	Demonstrates use of the serial driver, including graceful handling of errors.	
SIMPWP	Shows how to use a document control and display control to implement a simple word processor application. This is an excellent reference for using simple document and display groups.	
SUBUI	Demonstrates the subclassing of a Generic UI gadget. In this case, a new class of trigger has been created which changes its moniker when triggered.	
TICTAC	Demonstrates the manipulation of Vis objects, illustrated by a number of game pieces which the user may move with the mouse on the game board.	
TRIGGERD	Demonstrates how a simple trigger can be made much more versatile by fitting it with a trigger data area.	
TUTORIAL	This directory contains the application developed in the Tutorial book. The tutorial shows how the application was developed in several stages. Each subdirectory shows a stage of development.	
MCHRT1	A primary window.	
MCHRT2	Some Generic gadgetry.	
MCHRT3	Some graphics.	
MCHRT4	The Multiple Document Interface.	
TWODLIST	Demonstrates correct implementation of a common task—moving choices between two dynamic lists. Normally this is done when the user	

System Configuration

28

		is working with a number of things which may fall into two categories. They are presented with two lists, one for each category. Any list item they select will jump to the other category.
	UICTRL	Shows the use of a style control as used with a text object, along with a GenToolControl object. The user may hide or show the controller's tools and move them between two tool areas.
2.5	VARDATA	Demonstrates how to dynamically add and remove variable data fields to and from generic objects. In this case, the fields are geometry hints, so this application also shows the effects of some simple geometry hints on a pair of GenInteractions.
	VIEW	This directory contains a number of subdirectories, each of which contains a sample application showing some specialized usage of the GenView object. The most straightforward of these examples is viewsamp, and you might want to start with this one.
	GENCONTE	Demonstrates using a GenContent as a child of a GenView. This is a rather rare usage, and demonstrates some of the more powerful (if esoteric) uses of the Generic UI.
	SCALETOF	Demonstrates how to make a document's contents "scale to fit" when the view size changes.
	SPLITVIE	Demonstrates a split view—a view that has been set up to show more than one piece of a single document.
	IEWSAMP	Demonstrates some simple manipulations on a view, including scrolling, panning, and zooming.
	VIS	This directory contains a number of subdirectories, each of which in turn contains a sample application illustrating some facet of Visual objects and the Visual world.
	VISSAMP	General sample of vis objects running in a VisContent under a GenView. It includes examples of geometry management in a visible tree, a simple MSG_VIS_DRAW handler, basic mouse handling in a visible object, setting an object visible/not visible, addition/removal of objects, usage of VisMonikers, a simple MSG_VIS_RECALC_SIZE handler, custom positioning of objects, and marking an object invalid.
	VISSAMP2	Shows relation between a VisContent and its GenView in which the size of VisContent and bounds of its visible children



can be set in the object definitions without any geometry management or messages being sent to the objects whatsoever.

VISSAMP3	Demonstrates how to run a few objects under the content using the geometry manager. The view sizes itself, the content will use the view's size if possible, and the content will full justify its three Vis children horizontally while centering them vertically.	
VISSAMP4	Illustrates dynamically adding and removing objects to and from a Visible tree. Also shows geometry management of these objects and how it is affected when they are added or removed.	2.6
VTEXT	Demonstrates simple use of a visible text object.	
WAVSAMP	This sample application, meant for the Zoomer only, shows how to use the wav library to play sample sounds.	

2.6 Sample Esp Programs

The SDK_ASM directory contains a number of Esp sample programs. For the most part, these programs are Esp versions of previously described C sample programs. They are listed below.

HELLO	CLIPSAMP
FSELSAMP	SUBUI
DOCUMENT	UICTRL
HELLO2	GROBJ
LEVELS	TEXT
HELP	VOBJ
AVOID	

System Configuration

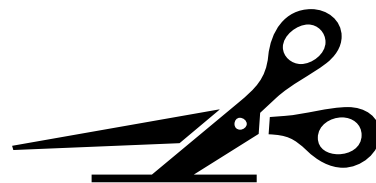
30

2.6



Swat

Introduction



3

3.1	DOS Command Line Options	35
3.2	Notation	36
3.3	Address Expressions	37
3.4	On-line Help.....	39
3.5	Essential Commands	42
3.5.1	Cycle of Development	42
3.5.2	Attaching and Detaching.....	43
3.5.3	Breakpoints and Code Stepping	47
3.5.3.1	Breakpoints	47
3.5.3.2	Code Stepping	52
3.5.4	Examining and Modifying Memory	54
3.5.4.1	Simple Memory Examination.....	54
3.5.4.2	Complex Memory Examination.....	57
3.5.4.3	Memory Examination with Modification.....	64
3.5.5	Other Important Commands	65
3.6	Additional Features	69





Most programmers are familiar with the process of debugging code. Many, however, will not be familiar with the issues of debugging programs in an entirely object-oriented, multithreaded system in which memory is often sharable by multiple programs. Because this type of system presents not only incredible power but also a new class of potential bugs, the GEOS tool kit provides a new class of symbolic debugger.

Swat is more than just a debugging program; it is an extensible debugging environment which you can tailor to your needs. Because all Swat commands are programmed in a special language called *Tool Command Language* (Tcl), you can write your own commands or extend existing commands. Tcl was originally developed by Professor John Ousterhout at the University of California in Berkeley, California. Swat itself was developed during a project headed by professor Ousterhout for a distributed multiprocessing operating system; Geoworks gained permission from the university to use and modify Swat. Since then, we have tailored it for use with GEOS; thus, it is the most appropriate debugger for any GEOS programmer.

Swat is essentially a system debugger, as opposed to an application debugger. This is an important distinction, due to the multithreaded nature of individual GEOS applications—each application may have more than one thread, and a system debugger greatly eases the debugging process for such applications. Swat also has many other features that make it preferable to other debuggers:

- ◆ **Ideal for multithreaded environment**
Because Swat was initially designed for debugging multithreaded system software, it is ideal for use on multithreaded applications in the GEOS environment.
- ◆ **Extremely flexible**
Nearly every part of Swat you will use is written in Tcl. Swat allows you to create your own commands or extend existing commands by using the Tcl language; you can examine, print, and modify just about any data structure using various Tcl commands. For large applications and projects, the customization this offers can be a tremendous asset.
- ◆ **Extensive access to data structures**
You can examine any byte in the test PC's memory while Swat is

attached. You can examine any basic or complex data structure. For example, the “pobject” command (“print object”) prints the entire instance data of the specified object. You can specify memory locations with symbols, registers, addresses, pointers, or a number of other ways.

- ◆ **Easy modification of code and data**
Using Tcl commands, you can change the contents of a register or memory location. You can also create patches to be executed at certain points in your program; this speeds up debugging by reducing the fix-compile-download-attach-debug cycle to a simple fix-debug cycle.
- ◆ **Interactive single-step facility**
By using Swat’s single-step command, you can execute a single instruction at a time. Swat shows you all the inputs going in to the instruction so you can more easily keep track of what’s going on.
- ◆ **Facilities for watching messages and objects**
Swat has several “watch” facilities. For example, you can let GEOS continue executing while watching a particular object; Swat will display all messages sent to the object, along with the data passed. You can also watch a particular message; Swat will display the destination and passed data each time the message is sent.
- ◆ **On-line help system**
Swat commands are documented on-line. The **help** facility is available from the basic Swat command prompt, and you can use it either as an interactive help shell or to get help on one particular topic. The **doc** function allows you to look up GEOS reference material in an ASCII version of the GEOS SDK technical documentation.

Swat also offers many other features that both novices and experts will use. As you gain familiarity with Swat while using Swat with the sample applications and with your own programs, you will discover your own preferred methods of debugging.



3.1 DOS Command Line Options

To use Swat, you must have the **pccom** tool running on the target machine. You may then invoke Swat on the development machine. Swat takes the following command-line flags:

- e** Start up the non-error-checking version of the loader on the target machine.
- h** Displays a usage line.
- k** Use a non-standard kernel file.
- n** Start up the non-error-checking version of GEOS on the target machine.
- r** Start up the error-checking version of GEOS on the target machine.
- s** Start up the error-checking loader on the target machine.
- D** You will only need to use this flag if debugging Swat.

3.1

If you have set up your communications incorrectly, Swat will have problems. (Often these problems don't show up when sending or receiving files; Swat demands more from the communications programs than pccom and pcget.)

One common problem arises when other devices are generating interrupts which are not being successfully masked out. If, for instance, you have a network card which is operating on IRQ 2, you must make sure that either the **pccom** tool or else Swat is called with the **/i:a** option. Swat will intercept IRQ level 5 by default. To determine what number to pass with the **/i**, take the IRQ level, add 8, and express the result in hexadecimal.

If Swat complains that it is "timing out" on some operation, you may have your communication speed set too high. Try changing the baud rate field in PTTY to a lower value.

Normally, Swat insists that any geodes it examines should have their symbolic information in the appropriate subdirectory of your root PCGEOS development directory—the possible subdirectories are Appl, Library, Driver, and Loader. To ask Swat to look in different places for these sorts of files, change the SWAT.CFG file in your PCGEOS\BIN directory. You may also specify a fifth directory in which to look for geodes. You may specify absolute

pathnames in this file; if you give relative paths, they will be assumed to start at the directory specified with your `ROOT_DIR` variable.

3.2 Notation

3.2

The rest of this chapter is devoted to interacting with Swat once you have it running. Most of this is done by means of commands typed at a prompt. Some Swat commands may have subcommands, some may have flag options, and some combine the two. Others may have special options; all, however, are documented with the following conventions.

- ◆ **command (alternative1 | alternative2 | ... | alternativeN)**
The parentheses enclose a set of alternatives separated by vertical lines (in C, the bitwise OR operator character). For example, the expression **quit (cont | leave)** means that either **quit cont** or **quit leave** can be used.
- ◆ **command <type_of_argument>**
The angled brackets enclose the type of an argument rather than the actual string to be typed. For example, **<addr>** indicates an address expression and **<argument>** indicates some sort of argument.
- ◆ **command [optional_argument]**
The brackets enclose optional arguments to the command. For example, the command **alias [<word>[<command>]]** could have zero, one, or two arguments because the *command* and *word* arguments are optional. Another example would be the command **objwalk [<addr>]**, which may take zero arguments if it is meant to use the default address or one argument if the user gives it a particular address to look at.
- ◆ * +
An asterisk following any of the previous constructs indicates zero or more repetitions of the construct may be typed. A plus sign indicates one or more repetitions of the construct may be used. For example, **unalias <word>*** can be the **unalias** command by itself, or it can be followed by a list of words to be unaliased.



3.3 Address Expressions

Address expressions are used as arguments to any Swat command that accesses memory. For example, the **pobject** command takes an address expression as an argument and prints out information about the object at that address. An address expression can be a symbol name, which is just the name of a pointer, or a *symbol path*. A symbol path looks like one of the following:

3.3

```
<patient>::
<module>::
<patient>::::<name>
```

The symbol path is used when there is more than one symbol of a given name or if a symbol of a different application is needed. A symbol can be represented in a variety of ways: the name of an object, a field of a structure, a register/number combination, a number from the address history, an element of an array, nested Tcl commands, or a Tcl variable. Array indexing is used as follows:

```
<addr> [<n>]
```

which will return the zero-based element *n* from the given *addr*, even if *addr* is not an array.

Another important way of representing the symbol is as a *segment:offset* pair. In this, the segment is a constant, a register, a module, or a handle ID given as **^h<id>** where *id* is a constant or register.

Some examples of address expressions are shown in Table 3-1.

Table 3-1 *Address Expressions*

Type	Example
name of object	Icon1
field of structure	applVars.Core
number/register combination	0x1ef0:si
number/register combination	1ef0h:si

There are several operators which are used to make memory examination and manipulation easier in Swat. These operators are shown below (in order of highest precedence to lowest):

◆ **^h**

The *carat-h* is used to dereference a memory handle when representing an address as a *handle:offset* pair (this is also known as a “heap pointer” representation) or when accessing a particular block of memory. It is often used in the situation when a memory handle is in one register (such as BX) and the offset is in another register (such as SI). This is similar to the ^l operator (below), but it requires an offset into the block rather than a chunk handle. The ^h operator is used thus (the two commands will give the same information if the specified registers contain the specified values):

```
[hello3:0] 6 => pobj ^h43d0h:0022h
[hello3:0] 7 => pobj ^hBX:SI
```

◆ **.**

The *period* is used to access a field in a structure. For example, if a visible object is located at ^hBX:SI, you could retrieve its top bound with the following command:

```
[hello3:0] 8 => print ^h43d0h:0022h.VI_bounds.R_top
```

◆ **+ -**

The addition and subtraction operators are used to add and subtract symbols to and from other symbols and constants. If two symbols in the same segment are subtracted, a constant will be the result.

◆ **^l**

The *carat-l* is used to dereference an optr, a pointer in the form *handle:chunk-handle* (this is also known as a “local memory pointer”). This is similar to the ^h operator, but ^l requires a chunk handle rather than an offset. If an optr is stored in CX:DX, for example, the ^l operator could be used to dereference it as follows:

```
[hello3:0] 11 => pobj ^lCX:DX
[hello3:0] 12 => pobj ^l0x43d0:0x022
```

◆ **:**

The *colon* is the segment/offset operator, used to separate the segment and offset in a *segment:offset* pair.

3.3



```
[hello3:0] 13 => pobj ^1CX:DX
[hello3:0] 14 => pobj ^10x43d0:0x022
[hello3:0] 15 => pobj INTERFACE:HelloView
```

◆ *

The *asterisk* is a pointer-dereferencing operator, as in the C programming language:

```
[hello3:0] 16 => print SubliminalTone
@5: SubliminalTone = 7246h
[hello3:0] 17 => print *(&SubliminalTone)
@6: *(&SubliminalTone) = 7246h
```

3.4

◆ ^v

The *carat-v* is the virtual memory operator, used to get to the base of a block that is in a Virtual Memory file given the file handle and VM block handle. The correct usage of the ^v operator is:

```
^v<file>:<VM_block>
```

Much of the time the type of data stored at the address given by the address expression is implicit in the expression. Sometimes in ambiguous situations (using code as data), however, the type of data must be explicitly stated in the address expression. This is done by indicating the type of the data followed by a space and then a normal address expression. For example, in the expression

```
dword ds:14h
```

the data at **ds:14h** will be treated as a double word.

3.4 On-line Help

Swat provides on-line help, both for looking up Swat topics and GEOS reference material.

To get help on a specific Swat command, you simply type the following, where the command is the argument.

```
[hello3:0] 7 => help <cmd>
```



Swat Introduction

40

To use Swat's interactive, menu-based help system, simply type the following:

```
[hello3:0] 8 => help
```

The menu-based system provides a tree of help topics which are organized into subjects. If you are looking for Swat commands having to do with a subject not covered in the help tree, you might try using the **apropos** command.

3.4

To get information about a GEOS topic, use the **doc** functions.

■ **apropos**

`apropos [<string>]`

The **apropos** command searches the list of commands and command help for all entries containing <string>. It lists each command and its synopsis. The string may actually be a partial word.

■ **doc, doc-next, doc-previous**

`doc [<keyword>]`

`doc-next`

`doc-previous`

The **doc** command looks for information in the technical documentation relevant to the passed keyword. The keyword may be any GEOS symbol. The **doc** command finds one or more places in the SDK technical documentation where the keyword is mentioned. It will display one place—to view the others, use the **doc-next** and **doc-previous** command. The documentation will appear in the source window. As when viewing source code using **srcwin**, you can scroll the view using the <Page Up> and <Page Down> and arrow keys.

■ **help**

`help [<command>]`

There are two different ways to use the help command. The first is to enter the *interactive help mode* using the **help** command with no arguments, and the second is to use the **help** command with a particular command as an argument.



Interactive help mode

The interactive help mode consists of a tree of commands and topics identified by different numbers. If one of the numbers is typed, information about that particular topic or command is displayed. Some of the topics have their own subtrees (indicated by the ellipses following the topic heading) which follow the same numbering format. The interactive help mode is used when looking for a certain style of command but the name of the command is not known (see Swat Display 3-1).

3.4

Swat Display 3-1 The Help Tree

```
(geos:0) 198 => help
```

top-most level of the help tree:

0 FINISH	6 memory...	12 step...
1 advanced...	7 object...	13 support...
2 breakpoint...	8 print...	14 swat_navigation...
3 crash...	9 running...	15 window...
4 file...	10 source...	
5 heap...	11 stack...	

Type "help" for help, "menu" to redisplay the menu, "0" to exit.

Type a topic (or its number) to display it.

```
help:top>
```

help <command>

When **help** is typed with another command as an argument, information about that command is displayed (the same information as in the interactive help mode). This command is frequently used in order to get fast help on a particular command. (See Swat Display 3-2.)

Swat Display 3-2 The help Command

```
(geos:0) 200 => help help
```

Help for help:

Functions for manipulating/accessing the help tree

```
=====
```

This is the user-level access to the on-line help facilities for Swat. If given a topic (e.g. "brk") as its argument, it will print all help strings



defined for the given topic (there could be more than one if the same name is used for both a variable and a procedure, for instance). If invoked without arguments, it will enter a browsing mode, allowing the user to work his/her way up and down the documentation tree

```
=====
(gEOS:0) 201 =>
```

3.5 Essential Commands

This section covers the function and usage of some of the most important Swat commands. These commands fall into the following command groups:

- ◆ **Cycle of Development**
Sending down a new copy of a geode and running it.
- ◆ **Attaching and Detaching Swat**
The commands used to control the link between Swat and GEOS.
- ◆ **Setting Breakpoints and Code Stepping**
The commands used to stop the execution of an application's code at pre-determined points and then examine the code, line by line if necessary.
- ◆ **Examination of Memory**
The commands used to examine memory from individual bytes to whole structures such as generic trees and objects.
- ◆ **Other Important Commands**
Other commands which are important to know but do not fit into the aforementioned groups.

A complete list of the Swat commands is contained in the Reference chapter.

3.5.1 Cycle of Development

`send, run, exit, patient-default`

These commands come in handy whenever you've edited and recompiled your application. You'll want to exit the application on the target machine. Use the



send command to send down the new, compiled version of your application. Then use the **run** command to start up the program.

■ send

```
send [<geode-name>]
```

To send the latest compiled version of your program, type “send” followed by the application’s patient name (the first part of the field on the “name” line of the .gp file).

3.5

■ run

```
run [<geode-name>]
```

To run your geode on the target machine, type “run” followed by the application’s patient name (the first part of the field on the “name” line of the .gp file).

■ exit

```
exit <geode-name>
```

To exit a running application, type “exit” followed by the application’s patient name. The exit command won’t work if your application has encountered a fatal error.

■ patient-default

```
patient-default [<geode-name>]
```

Use this command to set a default patient to use with the **send** and **run** commands. The send and run commands will operate on this patient if they are not passed arguments.

3.5.2 Attaching and Detaching

attach, att, detach, quit, cont, Ctrl-C

This group of commands controls the state of the connection between Swat and GEOS when Swat is running. The **attach** and **att** commands are used to establish the connection while **detach** and **quit** are used to sever it. The most frequently used commands in this group are **att** with the **-r** flag and

Swat Introduction

44

detach. Some related commands contained in the Reference are **go**, **istep**, **sstep**, and **next**.

The following is a typical but simplified debugging cycle using **detach** and **att**. It assumes you have already attached for the first time.

3.5

- ◆ When a bug is encountered, determine where and what the bug is, then detach Swat by typing **detach**.
- ◆ Edit the application to fix the bug, recompile it and download it to the target machine.
- ◆ Re-attach Swat to the target PC using the **att** command by typing Scrolllock-Shift-s on the target machine and typing **att** on the host machine.
- ◆ Continue debugging the application.
- ◆ Repeat the detach, edit, attach, debug cycle until all of the bugs are fixed.

By themselves, the commands shown below can not do much except open and close the communication lines between GEOS and Swat. More commands to examine and modify the application's code as it runs are needed to start actual debugging.

Swat Display 3-3 Detaching and Attaching

```
(geos:0) 202 => det cont
PC detached
(loader:0) 203 => att
Re-using patient geos
Re-using patient ms4
Re-using patient vidmem
Re-using patient swap
Re-using patient xms
Re-using patient disk
Re-using patient kbd
Re-using patient nimbus
Re-using patient stream
Re-using patient sound
Re-using patient standard
Re-using patient ui
Re-using patient styles
Re-using patient color
Re-using patient ruler
```



```

Re-using patient text
Re-using patient motif
Re-using patient vga
Re-using patient spool
Re-using patient serial
Re-using patient msSer
Re-using patient nonts
Re-using patient welcome
Re-using patient shell
Re-using patient manager
Re-using patient math
Re-using patient borlandc
Re-using patient messl
Thread 1 created for patient geos
Thread 2 created for patient geos
Thread 0 created for patient ui
Thread 0 created for patient spool
Thread 0 created for patient welcome
Thread 0 created for patient manager
Thread 1 created for patient manager
Thread 0 created for patient messl
Attached to PC
Stopped in 0070h:0005h, address 0070h:0005h
DOS+773: JMP DOS+2963
(geos:0) 204 =>

```

3.5

*In this example, we use the **det cont** command so that GEOS will keep running. We then re-attach with **att**. In the intervening time, the two machines are independent, and the serial line is unused. We could have taken advantage of this to send down a new copy of some application (as long as that application was not running on the target machine).*

■ att

att

The **att** command is similar to the **attach** command, but has no bootstrap argument (as explained below).

■ attach

attach [(+b|-b)]

This command is used to attach Swat to the target PC when the Swat stub is already invoked. The **-b** argument is to bootstrap and the **+b** argument is not to bootstrap. Bootstrapping means that Swat will search for the symbol files

of all of the geodes and threads as they are encountered rather than all at the beginning. This saves some time if you've just detached and need to re-attach, using only a few geodes while debugging. If no argument is given, the most recent bootstrap setting is used. The default bootstrap setting is **+b**.

■ **cont**

`cont`

3.5

The **cont** command continues the execution of GEOS after it has been stopped for some reason such as at a breakpoint or fatal error or by control-C. This command is often aliased as the letter **c**.

■ **detach**

`detach [(cont|leave)]`

The **detach** command will detach Swat from the target PC. By itself, the **detach** command will detach Swat and exit GEOS. This command is usually used after a bug is encountered and the source code needs to be modified and recompiled (see Swat Display 3-3).

cont The **cont** option will just detach Swat and allow GEOS to continue to run normally. This option is used when the debugging process is finished but GEOS is still needed to do other things (such as word processing) and you may need to re-attach later for further debugging.

leave The **leave** option will detach Swat but keep GEOS stopped wherever it was when the **detach leave** command was given. This command is useful for passing debugging control to someone remotely logged in to the workstation or when Swat can not continue for some reason.

■ **quit**

`quit [(cont|leave)]`

The **quit** command is only used when Swat needs to be exited for good. It will detach Swat (if necessary), exit from Swat on the development station, and exit from GEOS.

cont The **cont** option exits Swat on the development station but allows GEOS to continue running normally on the target PC. This option is used when the debugging process is finished but GEOS is still needed to do other things such as word processing.



leave The **leave** option will exit Swat but will keep GEOS stopped wherever it was when the **quit leave** command was given.

■ Ctrl-C

Ctrl-C

The **Control-C** command is the command used to stop the execution of GEOS at any point. This command is executed by holding down the **Ctrl** key and pressing the **c** key. It is used to stop GEOS in order to set a breakpoint, examine memory, or to get a command line prompt.

3.5

3.5.3 Breakpoints and Code Stepping

The commands in this group are used to stop at specified breakpoints in an application's code and then step through the code line by line if necessary. These commands are often used with each other to examine critical areas in the application source code.

3.5.3.1 Breakpoints

`stop, brk, go, cbrk, spawn`

The **stop**, **brk** and **cbrk** commands are used to set breakpoints. The breakpoint commands have many subcommands controlling the actions and attributes of a particular breakpoint.

The **cbrk** command sets breakpoints to be evaluated by the Swat stub; **brk** sets them to be evaluated by Swat. The Swat stub can evaluate the conditions much faster than Swat, but **cbrk** has certain limitations: only a limited number of breakpoints can be set using **cbrk**, and these breakpoints can only compare registers when the breakpoint is hit with a given set of criteria.

■ stop

```
stop in <class>::<message> [if <expr>]
stop in <procedure> [if <expr>]
stop in <address-history-token> [if <expr>]
stop at [<file:><line>] [if <expr>]
stop <address> [if <expr>]
```

3.5

This is the main command to use when setting breakpoints in C programs. The “stop in” command will set a breakpoint at the beginning of a procedure, immediately after the procedure’s stack frame has been set up. The “stop at” command will set a breakpoint at the first instruction of the given source line. If no <file> is specified, the source file for the current stack frame is used. If a condition is specified by means of an “if <expr>” clause, you should enclose the expression in curly braces to prevent any nested commands, such as a “value fetch” command, from being evaluated until the breakpoint is hit.

■ brk

```
brk [<sub-command>]
```

The **brk** (breakpoint) command is used for setting nearly all breakpoints in an application’s code. The simplest way to use it is to type **brk** with a single *addr* argument. The address is usually a routine name for a suspect procedure, and when the breakpoint is reached the code-stepping commands can be used to examine it carefully. The **brk** command can also create conditional breakpoints which will only be taken if certain conditions are satisfied. Once set, a breakpoint is given an integer number which can be obtained using the **list** subcommand (see Swat Display 3-4).

brk <addr> [<command>]

The **brk** command without any subcommands sets an unconditional breakpoint at the address specified in *addr*. If the *command* argument is passed, the given swat command will be carried out when the breakpoint is hit.

brk delete <break>*

Deletes the given breakpoint(s), just as **clear**, above.

brk enable <break>*

Enables the given breakpoint(s). Has no effect on previously enabled breakpoints. If no breakpoint is given, all breakpoints for the current patient are enabled.



brk disable <break>*

Disables the given breakpoint(s). It has no effect on previously disabled breakpoints. If no breakpoint is given, it disables all breakpoints for the current patient.

brk list [<addr>]

Lists all the breakpoints, whether they are enabled, where they are set, their conditions, and what actions they will take if encountered. If *addr* is given, it returns the breakpoint numbers of all breakpoints set at the given address.

3.5

Swat Display 3-4 Breakpoints

```
(geos:0) 4 => brk list
Num S Address Patient Command/Condition
1   E loader::kcode::LoaderError all echo Loader death due to [penum
                                LoaderStrings [read-reg ax]]
                                expr 1
2   E kcode::FatalError all
                                why
                                assign kdata::errorFlag 0
                                expr 1
3   E kcode::WarningNotice all why-warning
4   E kcode::CWARNINGNOTICE all why-warning
(geos:0) 5 => stop in Mess1Draw
brk5
(geos:0) 6 => brk list
Num S Address Patient Command/Condition
1   E loader::kcode::LoaderError all echo Loader death due to [penum
                                LoaderStrings [read-reg ax]]
                                expr 1
2   E kcode::FatalError all
                                why
                                assign kdata::errorFlag 0
                                expr 1
3   E kcode::WarningNotice all why-warning
4   E kcode::CWARNINGNOTICE all why-warning
5   E <ssl::MESS1_TEXT::Mess1Draw+10 all halt
(geos:0) 7 => brk dis brk4
(geos:0) 8 => brk list
Num S Address Patient Command/Condition
1   E loader::kcode::LoaderError all echo Loader death due to [penum
                                LoaderStrings [read-reg ax]]
                                expr 1
```



Swat Introduction

50

```
2  E kcode::FatalError all
                                why
                                assign kdata::errorFlag 0
                                expr 1
3  E kcode::WarningNotice all why-warning
4  D kcode::CWARNINGNOTICE all why-warning
5  E <ssl::MESS1_TEXT::Mess1Draw+10 all halt
(geos:0) 9 =>
```

3.5

■ go

go [<address-expressions>]

The **go** command sets a one-time breakpoint and resumes execution on the target PC. The net effect of this is to let the target go until it hits a given address, then stop.

■ cbrk

cbrk [<sub-command>]

The **cbrk** (conditional breakpoint) command is used to set fast conditional breakpoints. This command is very similar to the **brk** command above, except that the condition is evaluated by the Swat stub—this increases the speed of the evaluation. There are, however, certain restrictions on the **cbrk** command: only a limited number of breakpoints can be set (eight), and the scope of the evaluation is limited to comparing word registers (or a single word of memory) to a given set of values.

In the following descriptions, *criteria* stands for a series of one or more arguments of the form:

<register> <op> <value>

register One of the machine's registers or "thread," which corresponds to the current thread's handle.

op One of the following ten comparison operators: = (equal), != (not equal), > (unsigned greater-than), < (unsigned less-than), >= (unsigned greater-or-equal), <= (unsigned less-or-equal), +> (signed greater-than), +< (signed less-than), +>= (signed greater-or-equal), +<= (signed greater-or-equal). These correspond to the 8086 instructions JE, JNE, JA, JB, JAE, JBE, JG, JL, JGE, JLE, respectively.



value A standard Swat address expression. The resulting offset is the value with which the register will be compared when the breakpoint is hit.

cbrk <addr> <criteria>*

The basic cbrk command sets a fast conditional breakpoint at the address specified in *addr*.

cbrk cond <break> <criteria>*

Changes the criteria for the breakpoint. If no *criteria* is given the breakpoint becomes a standard, unconditional breakpoint.

3.5

■ **spawn**

```
spawn <patient-name> [<addr>]
```

The **spawn** command is used to set a temporary breakpoint in a process or thread which has not yet been created. The arguments are

patient-name

The permanent name, without extension, as specified by the name directive in the **.gp** file; this is the name of the patient in which to set a temporary breakpoint. A unique abbreviation is sufficient for this argument.

addr

A particular address at which to place the breakpoint. If no address is given, Swat will stop as soon as the given geode is loaded.

This command is used to stop the geode before any of its code can be run, allowing breakpoints to be set in the desired routines. If you could not stop the machine in this manner, the application could hit a buggy routine before a breakpoint could be set in that routine. The **spawn** command can also be used to catch the spawning of new threads which is useful to keep track of the threads being used by an application (see Swat Display 3-5).

Swat Display 3-5 The spawn Command

```
(geos:1) 12 => spawn mess1 Mess1Draw
Re-using patient math
Re-using patient borlandc
Re-using patient mess1
Thread 0 created for patient mess1
```



Swat Introduction

52

```
Interrupt 3: Breakpoint trap
Stopped in Mess1Draw, line 211, "C:\PCGEOS/Appl/SDK_C/MESS1/MESS1.GOC"
Mess1Draw(GStateHandle gstate) /* GState to draw to */
(mess1:0) 13 =>
```

3.5.3.2 Code Stepping

3.5

srcwin, istep, sstep

Once an application is stopped at a breakpoint and you want to examine the code line by line, you can use the commands **istep** (instruction step) and **sstep** (source step). These enter the *instruction step mode* or *source step mode* to examine and execute the application code line by line.

The subcommands for both **istep** and **sstep** are nearly the same and are used for actions including stepping to the next line, skipping the next instruction, or exiting the step mode and continuing the execution of the application. The **istep** and **sstep** commands are very similar except that **istep** is used when stepping through assembly source code (thus stepping through instructions), and **sstep** is used for stepping through C source code.

■ srcwin

srcwin <numlines> [view]

The srcwin command will display the source code surrounding the presently executing code any time execution is stopped. The presently executing line will be highlighted. You may set breakpoints with the mouse by clicking on the line numbers which appear to the side. To scroll the srcwin buffer use the arrow keys, the <PgUp> key, and the <PgDn> key.

■ istep, sstep

istep [<default subcommand>]

sstep [<default subcommand>]

These two commands are used to single-step through code, executing one or more instructions at a time. The *default subcommand* argument determines the action taken by Swat when the Return key is pressed. For example, the command

```
[hello3:0] 7 => istep n
```



will enter instruction step mode, and subsequently pressing the Return key will have the same effect as pressing **n**. If no default command is given, pressing Return has the same effect as pressing **s**.

The subcommands to the **istep** and **sstep** commands are

s (single step)

Step one instruction. This is the most frequently used subcommand.

n, o (next, over)

Continue to the next instruction but do not display any procedure calls, repeated string instructions, or software interrupts. They will stop when GEOS returns to the same frame as the previous displayed instruction. The frame is the same when the stack pointer and current thread are the same as when the **n** subcommand was given. **o** differs from **n** in that it executes all instructions in a macro without further interpretation and can only be used with **istep**. If a breakpoint other than for the next instruction is hit, it will take effect as long as the above conditions are met.

3.5

N, O (Next, Over)

These are like **n** and **o** but will stop whenever a breakpoint is hit even if the frame is different. **O** will execute all instructions in a macro without further interpretation, and it can only be used with **istep**. If a breakpoint other than one for the next instruction is hit, it will take effect as long as the above conditions are met.

q, Esc, <space> (quit)

These stop **istep/sstep** and return to the command level. These subcommands are used when a point in the code is reached where another command needs to be used—to examine memory, for example.

c (continue)

This exits **istep** and continues the execution of the application. When GEOS next stops, Swat will return to the command prompt.

M (message)

This will continue until the next handled message is received. When the handler is invoked, Swat will return to step mode. This subcommand is often used with the **ObjMessage()** and **ObjCallInstanceNoLock()** assembly routines.

F (finish message)

This finishes the current message, stops when execution returns to a frame that is not part of the kernel, and remains in step mode.



f (finish frame)

This finishes the current stack frame, stops, and remains in step mode.

S (skip instruction)

This skips the current instruction, does not execute it, and goes on to the next instruction in step mode.

3.5

3.5.4 Examining and Modifying Memory

The commands in this section all deal with the examination, manipulation, or modification of the memory used by an application. Memory from individual bytes to complex data structures such as objects can be displayed and examined. These commands fall into the following groups:

- ◆ **Simple Memory Examination.**
Examination of bytes, words, and double words with no modification.
- ◆ **Complex Memory Examination**
Examination of structures such as objects, generic trees, and handle tables with no modification.
- ◆ **Memory Examination with Modification**
Examination of memory with modification if desired. Some commands are used only for memory modification.

The commands in this section are often used with each other and with the code-stepping and breakpoint commands in order to pinpoint bugs in an application's code. Breakpoints can be set, code can be stepped through, and then the memory that the code uses can be examined.

Some related commands defined in the Reference chapter are **down**, **func**, **handles**, **hgwalk**, **impliedgrab**, **penum**, **phandle**, **pinst**, **piv**, **precord**, **skip**, **systemobj**, **up**, and **where**.

3.5.4.1 Simple Memory Examination

`bytes`, `words`, `dwords`, `frame`, `backtrace`, `why`, `listi`

The commands in this group are used to look at simple blocks of memory without modification. They are defined fully in the entries below.



■ bytes, words, dwords

```
bytes [<addr>] [<length>]
words [<addr>] [<length>]
dwords [<addr>] [<length>]
```

The **bytes**, **words**, and **dwords** commands are essentially the same except that each looks at a different sized piece of memory. These commands will display the memory as a pointer to a dump of bytes, words, or dwords using the given or most recent (if no address is given) address.

3.5

The **bytes** command additionally displays the dump as its ASCII character representation, if any. These three commands are used to examine memory on a very basic level and are useful only if the user knows what the bytes, words, or dwords should or should not be and so can spot any problems. For example, if a certain character string such as “Application” is supposed to be stored at the address given by *fileType* and the command

```
[hello3:0] 11 => bytes fileType
```

dumps the characters “noitacilppA”, then there is most likely a problem.

These commands will automatically use the given *addr* as a pointer to the memory to be examined. If Return is hit many times in a row, the result will be to examine adjacent pieces of memory. (See Swat Display 3-6.)

Swat Display 3-6 The words Command

```
(mess1:0) 15 => words themeSongBuf
Addr: +0 +2 +4 +6 +8 +a +c +e
0040h: 0004 0000 0049 0000 0004 0001 0083 0000
(mess1:0) 16 => words
Addr: +0 +2 +4 +6 +8 +a +c +e
004eh: 0000 0006 0004 0028 0000 0001 020b b800
(mess1:0) 17 => !!
words
Addr: +0 +2 +4 +6 +8 +a +c +e
005ch: b800 000c 0020 0002 0001 0000 0001 020b
```

Swat Introduction

56

```
(mess1:0) 18 => !!
words
Addr: +0 +2 +4 +6 +8 +a +c +e
006ah: 020b b800 000c 0010 0002 0001 000a 0010
```

Here the **words** command examines a buffer in memory. When this command is repeated without arguments, it will display memory continuing where the last command left off. Note the use of the **!!** command to repeat the previous command.

3.5

■ backtrace, frame

```
backtrace [<frames to list>]
frame <subcommand>
```

The **backtrace** and **frame** commands are used to examine data that has been pushed onto the stack. An application may crash in a routine that is correctly written but has been passed bad data.

The **backtrace** command prints out a list of all the active frames for the current patient. Then the user can choose a particular frame to examine using one of the **frame** subcommands. The **frame** command is used to access frames that have been pushed onto the stack, where a *frame* is the information for a routine that needs to be saved when it calls another routine.

The **frame** and **backtrace** commands can be used together to print the active frames with **backtrace** and then access data in these frames with **frame**. However, most of the **frame** subcommands expect a token for a frame, not the frame number given by the backtrace command. To get this token, the **top**, **cur** and **next** subcommands are used. Then the other **frame** subcommands can be used with the token to further examine the **frame** data (see Swat Display 3-7). See “Swat Reference,” Chapter 4, for more details on the **frame** command.

Swat Display 3-7 Backtrace and Frame commands

```
Death due to SOUND_BAD_EVENT_COMMAND
Execution died in patient sound:
SoundHandleTimerEvent+63: MOV AX, 7 (0007h)
*** No explanation available ***
Interrupt 3: Breakpoint trap
```



```

Stopped in FatalError, address 1844h:0163h
SoundHandleTimerEvent+63: MOV AX, 7 (0007h)
(mess1:0) 2 => backtrace
  1: near FatalError(), 1844h:0163h
  2: far AppFatalError(), 1844h:0163h
* 3: far SoundHandleTimerEvent(), 2cb2h:003fh
  4: far SoundLibDriverPlaySimpleFM(), 6247h:0062h
  5: far ResourceCallInt(), 1844h:1492h
  6: far SoundLibDriverStrategy(), 2cb2h:0ab2h
  7: near SoundCallLibraryDriverRoutine(), 629ch:00feh
  8: far SoundPlayMusic(), 629ch:0028h
  9: far ResourceCallInt(), 1844h:1492h
 10: far SOUNDPLAYMUSICNOTE(mh = ^h42b0h (at 753ch), priority = 1eh, tempo = 4h
, flags = 80h), 62d6h:00f3h
 11: far ResourceCallInt(), 1844h:1492h
 12: far Mess1Draw(), MESS1.GOC:307
 13: far MESS1PROCESSMETA_EXPOSED(win = 3a60h, message = 69 (invalid), oself =
3ee0h:0000h), MESS1.GOC:362
 14: far ResourceCallInt(), 1844h:1492h
MSG_META_EXPOSED (3a60h 0000h 0000h) sent to Mess1ProcessClass (^l20f0h:0h)
 16: near ObjCallMethodTableSaveBXSI(), 1844h:9ea5h
 17: far SendMessage(), 1844h:9d9bh
 18: far ObjMessage(), 1844h:1d9ch
 19: far MessageDispatchDefaultCallback(), 1844h:1c72h
 20: far MessageProcess(callBack = 1844h:1c68h (geos::kcode::MessageDispatchDef
aultCallback)), 1844h:1c15h
 21: far MessageDispatch(), 1844h:1b31h
 22: far ThreadAttachToQueue(), 1844h:bd2ch
(mess1:0) 3 => frame 12
Mess1Draw+302: MOV AX, 100 (0064h)

```

3.5

3.5.4.2 Complex Memory Examination

print, hwalk, lhwalk, objwalk, pobject, gentree, vistree,
vup, gup

The commands in this group are used to examine complex data structures in GEOS.



■ print

```
print <expression>
```

The **print** command is used to print out the value of the given *expression* argument. The *expression* argument is normally some sort of typed address. When there is no type for the *expression*, then its offset is printed.

3.5

The power of this command lies in its ability to print any type at any address; thus, it is used frequently to print out the values of important expressions such as registers or variables. The **print** command also takes many flags which control the way in which the value of the *expression* is displayed such as in decimal or hexadecimal. See the Reference chapter for more information on the flags for the **print** command.

■ hwalk

```
hwalk [<patient>]
```

Use the **hwalk** (heap walk) command to display blocks on the global heap. Its output can be tailored in various ways according to how the *flags* are set. If a *patient* is given, then **hwalk** will only print the blocks owned by that patient. There are many fields in the listing such as the handle, address, and type of each block. By examining these fields, the user can get an overall sense of how the global heap is being managed, whether any block looks too big or too small, and what the handles of important blocks are. (See Swat Display 3-8.)

Swat Display 3-8 The hwalk Command

```
(mess1:0) 6 => hwalk mess1
```

HANDLE	ADDR	SIZE	FLAGS	LOCK	OWNER	IDLE	OINFO	TYPE
20f0h	41ebh	2272	FIXED	n/a	mess1	n/a	1h	R#1 (dgroup)
4160h	58eah	448	sDS	a	1 mess1	105eh	1h	R#2 (MESS1_TEXT)
3a60h	59adh	784	s SL	0	mess1	0:03	1h	WINDOW
4bb0h	6176h	560	s SL	0	mess1	0:05	1h	WINDOW
3970h	6232h	336	s SL	0	mess1	0:03	1h	GSTATE
3ee0h	633ch	160	s S	a	0 mess1	0:05	49c0h	Geode
4950h	63beh	1280	s SL	0	mess1	0:05	49c0h	OBJ(mess1:0)
4340h	640eh	1328	SL	0	mess1	0:05	49c0h	R#3 (INTERFACE)
42b0h	753ch	96	s S	4	mess1	1249h	1h	
4bd0h	7542h	96	s S	0	mess1	0:01	1h	



```
41b0h 89d4h 896 SL 0 mess1 0:05 49c0h R#4 (APPRESOURCE)
4270h 99e1h 32 s S 0 mess1 0:05 1h
```

```
Total bytes allocated: 8288
(mess1:0) 7 =>
```

■ lhwalk, objwalk

```
lhwalk [<addr>]
objwalk [<addr>]
```

3.5

The **lhwalk** (local heap walk) command is used to display information about a local memory heap, and the **objwalk** command is used to print out information about an object block. After using **hwalk** to locate a specific block, **lhwalk** or **objwalk** can be used to print out information about that particular block. These commands also print out fields of information which include the local handle, the address, size, and type of data or object. See the Reference chapter for more information on the fields printed by **lhwalk** and **objwalk**. (See Swat Display 3-9.)

Swat Display 3-9 The objwalk Command

```
(mess1:0) 11 => objwalk ^h4340h
```

```
Heap at 640eh:0 (^h4340h), Type = LMEM_TYPE_OBJ_BLOCK
In use count = 3, Block size = 1328, Resource size = 59647 para (192 bytes)
```

HANDLE	ADDRESS	SIZE	FLAGS	CLASS	(NAME)
001ch	56h	1eh	---	*flags*	
001eh	76h	c1h	D RO	GenPrimaryClass	(Mess1Primary)
0020h	1aah	ceh	D RO	GenViewClass	(Mess1View)
0022h	166h	32h	ID O	OLGadgetAreaClass	
0024h	19ah	eh	I		
0026h	492h	6bh	D O	GenValueClass	
0028h	2deh	6bh	D O	GenValueClass	
002ah	37eh	a6h	ID O	GenInteractionClass	
002ch	44ah	46h	ID O	OLMenuBarClass	
002eh	13ah	bh	ID		
0030h	426h	22h	ID O	OLMenuButtonClass	



Swat Introduction

60

```
Free handles = 17, null handles = 0
Objects = 8, 4 of them marked ignoreDirty

(mess1:0) 12 =>
```

■ pobject

3.5

```
pobject [<addr>] [<print level>]
```

The **pobject** (print object) command (often abbreviated **pobj**) is used to print out the entire instance data chunk of an object. You can use **gentree**, **vistree**, or **hwalk** and **objwalk** to get the handles for an object; once you have them, use **pobj** with the handles, as follows:

```
[hello3:0] 7 => pobj ^10x43d0:0x0022
```

will print out the instance data chunk specified by that optr.

Any valid address expression, such as a dereferenced object name, may be used as an *addr*. Additionally, the print level can be changed to print just the headings to each of the master levels and an address history number. The **pobject** command is used to verify that the object is behaving correctly and that its instance variables (if any) are correct. (See Swat Display 3-10.)

Swat Display 3-10 The pobject Command

```
[lesink:0] 10 => pobj ^14710h:0020h
*UpTextView::UpTextViewClass (@7, ^14710h:0020h)
master part: Gen_offset(123) -- UpTextViewInstance
@8: {UpTextViewInstance (^h18192:622)+123} = {
  MetaBase Gen = {
    ClassStruct _far *MB_class = 360ah:162fh (motif::dgroup::OLPaneClass)
  }
  LinkPart GI_link = {
    dword LP_next = 4710h:001fh
  }
  CompPart GI_comp = {
    dword CP_firstChild = 4710h:002ah
  }
  word GI_visMoniker = 0h
  word GI_kbdAccelerator = 0h
  byte GI_attrs = 2h
  byte GI_states = c0h
}
```



```

PointDWord GVI_origin = {
    DWord PDF_x = {0.000000}
    DWord PDF_y = {0.000000}
}
RectDWord GVI_docBounds = {
    long RD_left = 0
    long RD_top = 0
    long RD_right = +480
    long RD_bottom = +480
}
PointDWord GVI_increment = {
    long PD_x = +20
    long PD_y = +15
}
PointWord GVI_scaleFactor = {
    Word PF_x = {1.000000}
    Word PF_y = {1.000000}
}
ColorQuad GVI_color = {
    CQ_redOrIndex = fh, CQ_info = 0h, CQ_green = 0h, CQ_blue = 0h
}
word GVI_attrs = 10h
byte GVI_horizAttrs = 88h
byte GVI_vertAttrs = 88h
byte GVI_inkType = 0h
dword GVI_content = 4710h:0024h
dword GVI_horizLink = 0000h:0000h
dword GVI_vertLink = 0000h:0000h
}
Variable Data:
    *** No Variable Data ***
[lesink:0] 11 =>

```

3.5

In addition to printing information about the object at a given address, pobject can print information about certain objects in the application if passed certain flags:

- pobject -i** Prints information about the windowed object under the mouse pointer.
- pobject -c** Prints information about the content for the view over which the mouse is located.



There are more flags available, and it is also possible to ask for more or less instance data information. See the full reference for this command for details.

■ gentree

gentree [<addr>] [<instance field>]

3.5

The **gentree** (generic tree) command prints out a generic tree from the given *addr* and *instance field*. The *addr* must be the address of an object in the generic tree, and the *instance field* must be the offset into the Generic master part of the instance chunk or any instance data within the Generic master level which is to be printed. This command is used primarily to ensure correct structure of a generic tree and its instance data and to find a particular object in the generic tree. The **-i** (implied grab) option is used to find an object by placing the mouse over the window in which the object resides and typing the following:

```
[hello3:0] 7 => gentree -i
```

The default address that **gentree** examines is contained in *DS:SI. (See Swat Display 3-11.) To examine objects more closely, pass the handles displayed by **gentree** to the **pobject** command.

■ vistree

vistree [<addr>] [<instance field>]

The **vistree** (visual tree) command prints out a visual tree from the given *addr* and *instance field*. The *addr* must be the address of an object in the visual tree, and the *instance field* must be the offset into the Vis master part of the object's instance data which is to be printed. This command is primarily used to examine the on-screen layout of the application and to ensure correct structure of the visual tree and its instance data. The **vistree** command can use the **-i** option (implied grab), which will use the window that the mouse is over as the first visual object in the printed tree. The default address that **vistree** examines is contained in *DS:DI. To examine objects more closely, pass the handles displayed by **vistree** to the **pobject** command.



■ gup

gup [<addr>] [<instance field>]

The **gup** (Generic UPward query) command is used to go up the generic tree from a particular object specified by the *addr* argument, the default *DS:SI, or the **-i** option. The **-i** option (implied grab) uses the windowed object under the mouse as the object from which to start the upward query. This command is used primarily to ensure correct generic class hierarchy and to determine the field of the given object.

3.5

Swat Display 3-11 Gentree and Gup

```
(mess1:0) 19 => gentree -i
```

```
GenViewClass (@5, ^14340h:0020h)
GenValueClass (@6, ^14340h:0026h)
GenValueClass (@7, ^14340h:0028h)
```

```
(mess1:0) 20 => gup @5
```

```
GenViewClass (@11, ^14340h:0020h)
GenPrimaryClass (@12, ^14340h:001eh) "MESS #1"
GenApplicationClass (@13, ^141b0h:0024h) *** Is Moniker List ***
GenFieldClass (@14, ^14080h:001eh)
GenSystemClass (@15, ^12460h:0020h)
```

```
(mess1:0) 21 => gentree ^14340h:001eh
```

```
GenPrimaryClass (@16, ^14340h:001eh) "MESS #1"
GenViewClass (@17, ^14340h:0020h)
GenValueClass (@18, ^14340h:0026h)
GenValueClass (@19, ^14340h:0028h)
```

■ vup

vup [<addr>] [<instance field>]

The **vup** (Visual UPward query) command is used to examine the visual ancestors of a particular object given by the *addr* argument, the default *DS:SI, or the **-i** option. The **vup** command can be used with the **-i** option (implied grab) to use the windowed object under the mouse as the object from



which to start the upward query. This command is used primarily to ensure correct visual class hierarchy and to determine the field of the given object.

3.5.4.3 Memory Examination with Modification

`assign, imem`

3.5

The commands in this group are used to modify memory without detaching Swat and editing the application code. They are often used in conjunction with **istep**, **sstep**, and **pobject** to fix any small errors while the code is executing rather than detaching, modifying the actual code, and recompiling. These fixes are temporary, and you must change the source code to enact the real bug fixes.

■ **assign**

`assign <addr> <value>`

The **assign** command will assign the given *value* to the given *addr*, which can only have type **byte**, **word**, or **dword**. Both memory locations and registers may be assigned new values. This command is used to correct minor mistakes or test differing values at run-time without having to recompile.

■ **imem**

`imem [<addr>] [<mode>]`

The **imem** (inspect memory) command combines examination and modification of memory into one command. It can be used to search through areas of memory and modify problem areas selectively. The command is used to print out memory starting at either the given *addr* or at the default DS:SI in one of the following modes:

- b** (bytes) Displays the memory in terms of bytes.
- w** (words) Displays the memory in terms of words.
- d** (double words)
 Displays the memory in terms of double words.
- i** (instructions)
 Displays the memory in terms of instructions.

There are many subcommands to **imem** which are executed in the same manner as those for **istep** and **sstep**. These subcommands are as follows:



- b, w, d, i** These will reset the mode to the given letter and redisplay the data in that mode.
- n, j, <return>** (next, jump)
This will advance to the next piece of data using the appropriate step size (dependent upon the display mode).
- p, k, P** (previous)
This will retreat to the preceding piece of data. While in instruction mode, if the displayed instruction is wrong, try again with the **P** subcommand.
- <space>** This will clear the data being displayed and allow you to enter a new value in accordance with the current mode. This is exactly like the **assign** command except for singly and doubly quoted strings. A singly quoted string such as 'hello' will have its characters entered into memory starting at the current address with no null byte at the end. A doubly quoted string such as "good-bye" will be entered into memory at the current address with the addition of a null byte at the end of the string. This subcommand may not be used in instruction mode.
- q** (quit) Quits the **imem** mode and returns to the command level.
- Ctrl-d** Control-d (down) displays ten successive memory elements in the current display mode.
- Ctrl-u** Control-u (up) displays ten of the preceding memory elements in the current display mode.

3.5

3.5.5 Other Important Commands

alias, mwatch, objwatch, save, switch, sym-default, why

These commands are important to know but do not readily fall into any of the previous categories. This section will discuss each of these commands in relation to the debugging process.

■ alias

alias [<name> [<body>]]

This command is normally used to abbreviate a long command or series of commands with one single, descriptive command. If no arguments are given, then **alias** will just give a list of all the aliases and the commands they alias.



Swat Introduction

66

The **alias** command is a convenient shortcut for oft used commands or for commands that take a long time to type.

If only one argument is given, then **alias** will try to match that argument to the command it is aliased to. For example, if the **print** command is aliased to **p**, then **alias p** will return **print** as a result. If two arguments are given, then **alias** will cause *argument1* to be allowed as an alternative to typing *argument2*. For example, if the command **print** were to be aliased as **p**, the **alias** command would be used as below:

3.5

```
[hello:0] 5 => alias p print
```

Typing **p** will now have the same effect in Swat as typing **print**.

■ mwatch

```
mwatch [<message>+]
```

The **mwatch** (message watch) command watches a particular message and displays all deliveries of that message without stopping GEOS. This command can help to verify that a particular message is getting sent to all the right places and is not sent to any of the wrong places. Up to eight messages can be watched at once, and the **mwatch** command with no arguments clears all watched messages. Note that some message handlers will relay a message on to a superclass' handler; this may make it appear that the message is being delivered again, though this is not the case. (See Swat Display 3-12.)

Swat Display 3-12 The mwatch Command

```
(ui:0) 30 => mwatch MSG_VIS_DRAW MSG_META_EXPOSED
(ui:0) 31 => c
MSG_VIS_DRAW, ^12860h:001eh, GenInteractionClass
  cx = 3f80h, dx = 0000h, bp = 3950h
MSG_VIS_DRAW, ^12860h:001eh, GenInteractionClass
  cx = 3f80h, dx = 0000h, bp = 3950h
MSG_VIS_DRAW, ^12860h:001eh, GenInteractionClass
  cx = 3f80h, dx = 0016h, bp = 3950h
MSG_VIS_DRAW, ^12860h:001eh, GenInteractionClass
  cx = 3f80h, dx = 0016h, bp = 3950h
MSG_VIS_DRAW, ^12860h:002ch, OLGadgetAreaClass
  cx = 3f80h, dx = 3950h, bp = 3950h
MSG_VIS_DRAW, ^12860h:002ch, OLGadgetAreaClass
  cx = 3f80h, dx = 3950h, bp = 3950h
```



```
MSG_VIS_DRAW, ^12860h:002ch, OLGadgetAreaClass
  cx = 3f80h, dx = 007ch, bp = 3950h
MSG_VIS_DRAW, ^12860h:002ch, OLGadgetAreaClass
  cx = 3f80h, dx = 007ch, bp = 3950h
MSG_VIS_DRAW, ^12860h:0020h, GenTextClass
  cx = 3f80h, dx = 3950h, bp = 3950h
MSG_VIS_DRAW, ^12860h:0020h, GenTextClass
  cx = 3f80h, dx = 3950h, bp = 3950h
MSG_VIS_DRAW, ^12860h:0020h, GenTextClass
  cx = 3f80h, dx = 3950h, bp = 3950h
```

3.5

■ objwatch

objwatch [<addr>]

The **objwatch** (object watch) command is used for displaying the messages that have reached a particular object. It is useful for verifying that messages are being sent to the object at *addr*. If no argument is given, then any current **objwatch** is turned off. (See Swat Display 3-13.)

Swat Display 3-13 The objwatch Command

```
mess1:0) 2 => objwatch Mess1View
brk5
(mess1:0) 3 => c
MSG_META_MOUSE_PTR, ^144a0h:0020h, GenViewClass
  cx = 00afh, dx = 0013h, bp = 0000h
MSG_META_MOUSE_PTR, ^144a0h:0020h, GenViewClass
  cx = 00afh, dx = 0013h, bp = 0000h
MSG_META_WIN_UPDATE_COMPLETE, ^144a0h:0020h, GenViewClass
  cx = 4b90h, dx = 0000h, bp = 0000h
MSG_META_MOUSE_PTR, ^144a0h:0020h, GenViewClass
  cx = 00b0h, dx = 0013h, bp = 0000h
MSG_META_RAW_UNIV_LEAVE, ^144a0h:0020h, GenViewClass
  cx = 44a0h, dx = 0020h, bp = 4b90h
MSG_VIS_DRAW, ^144a0h:0020h, GenViewClass
  cx = 4b80h, dx = 23c0h, bp = 23c0h
MSG_VIS_COMP_GET_MARGINS, ^144a0h:0020h, GenViewClass
  cx = 4b80h, dx = 23c0h, bp = 0000h
```



Swat Introduction

68

```
MSG_VIS_DRAW, ^144a0h:0020h, GenViewClass
cx = 4b80h, dx = 0163h, bp = 23c0h
MSG_META_WIN_UPDATE_COMPLETE, ^144a0h:0020h, GenViewClass
cx = 4b90h, dx = 0000h, bp = 0000h
```

■ save

3.5

```
save <filename>
```

The **save** command, when passed a file name, saves the contents of Swat's main buffer to that file. Thus this command dumps Swat output to a file.

■ switch

```
switch [(<patient>:<thread-num>|<threadID>)]
```

The **switch** command is used to switch between applications or threads in Swat but does not physically change threads on the target PC. This allows the transfer of debugging control between threads of the same patient. If no argument is given, then **switch** will change to the thread executing when GEOS was halted. Another way to switch threads is to type the name of the patient on the command line. If a patient has more than one thread, type the name of the patient, a space, and then the thread number. To change thread numbers within a geode, type a colon followed by the thread number to change to (e.g. ":1")

■ sym-default

```
sym-default [<patient>]
```

The **sym-default** (symbol default) command is used to set the default patient to use when parsing an address expression which is not defined in the current patient. For example, if a breakpoint is hit in the kernel and an object in the application code needs to be examined, Swat will know to use the application as a patient and not the kernel. This command is useful when debugging a single patient, the most common way to debug. If no *patient* argument is given, then the name of the default patient will be displayed.

This command is normally aliased to **sd**.



■ why

why

The **why** command prints the error code for an occurrence of a fatal error. This command is useful because it can give a good idea of why GEOS crashed.

3.6 Additional Features

3.6

This section covers the features of Swat that make it easier to use when debugging an application.

◆ Mouse support

You can use the mouse to capture and paste text in the main Swat buffer. Capture text by click-dragging with the left mouse button. Pressing the right mouse button pastes the captured text to the Swat prompt line.

◆ Navigating the Main Buffer

To scroll the main buffer, use Ctrl-u (up), Ctrl-d (down), Ctrl-y (back one line), Ctrl-e (forward one line), Ctrl-b (backward page) and Ctrl-f (forward page).

◆ Command History

By pressing Ctrl-p several times, you can call previous commands up to the Swat prompt. If you go past the command that you want, use Ctrl-n to go forward in the history.

The '!' character followed by a number repeats that command in the command history. (The standard Swat prompt includes a command number which may be used for this.) e.g. !184 will execute the 184th command of this session.

The '!' character followed by a string will repeat the most recent command whose beginning is the same as the passed string. That is !b might invoke **brk list** if that was the most recent command that began with "b".

Typing "!!" will repeat the previous command; " !\$" is the last argument of the previous command.

◆ Command Correction

To repeat the previous command, but changing a piece of it, use the ^ command. This comes in handy when you've made a typo trying to enter the previous command.

Swat Introduction

70

Swat Display 3-14 Command Correction Using ^

```
(geos:0) 185 => wurds  
Error: invoked "wurds", which isn't a valid command name
```

```
(geos:0) 186 => ^u^o  
words  
Addr:          +0   +2   +4   +6   +8   +a   +c   +e  
4b4bh: e800 01b1 0e00 60f6 0016 9800 6e02 a900
```

3.6

```
(geos:0) 187 => ddwords  
Error: invoked "ddwords", which isn't a valid command name
```

```
(geos:0) 188 => ^d  
dwords  
Addr:  +0          +4          +8          +c  
4b59h: 1d0aa900 001c400d 294bd000 6c0a8000
```

◆ Address History

Swat has an address history which is composed of tokens for address expressions previously used by commands such as **print** or **pobj**. The elements in the history can be accessed by typing **@<number>** where the *number* argument is the number of the item in the history. These elements can replace a full address expression (except constants) and are often used when traversing through fields of a previously printed structure. The default history keeps track of the last 50 items. (See Swat Display 3-15.)

Swat Display 3-15 The Address History

```
(geos:0) 8 => gentree -i
```

```
GenPrimaryClass (@1, ^144a0h:001eh) "MESS #1"  
  GenViewClass (@2, ^144a0h:0020h)  
    GenValueClass (@3, ^144a0h:0026h)  
      GenValueClass (@4, ^144a0h:0028h)
```

```
(geos:0) 9 => pinst @3  
class = ui::dgroup::GenValueClass  
master part: Gen_offset(53) -- ui::GenValueInstance  
@5: {ui::GenValueInstance (^h17568:1170)+53} = {  
  GenInstance GenValue_metaInstance = {  
    MetaBase Gen_metaInstance = {
```



```

ClassStruct _far *MB_class = 3573h:1867h (motif::dgroup::OLScrollbarClass)
}
LinkPart GI_link = {
void _optr LP_next = ^144a0h:0028h (ui::dgroup::GenValueClass@6244h:02deh)
}
CompPart GI_comp = {
void _optr CP_firstChild = null
}
void _lptr GI_visMoniker = null
KeyboardShortcut GI_kbdAccelerator = {
KS_PHYSICAL = 0
KS_ALT = 0
KS_CTRL = 0
KS_SHIFT = 0
KS_CHAR_SET = 0
KS_CHAR = C_NULL
}
GenAttrs GI_attrs = {}
GenStates GI_states = {GS_USABLE, GS_ENABLED}
}
WWFixed GVLI_value = {0.000000}
WWFixed GVLI_minimum = {0.000000}
WWFixed GVLI_maximum = {0.007324}
WWFixed GVLI_increment = {0.000229}
GenValueStateFlags GVLI_stateFlags = {GVSF_INDETERMINATE}
GenValueDisplayFormat GVLI_displayFormat = GVDF_INTEGER
void _optr GVLI_destination = ^144a0h:0020h (ui::dgroup::GenViewClass@6244h:01aah)
word GVLI_applyMsg = 681ah
}
}
(geos:0) 10 =>

```

3.6

◆ Abbreviations

Another shortcut available in Swat is the abbreviation feature. Many commands can be specified by their first few characters up to and including the letter that makes them distinct from all other commands. For example, the **pobject** command can be specified **pobj**, **pob**, or even **po**, but not by just **p** because there are other commands (such as **print**) beginning with the letter **p**. To get a list of all commands with a given prefix, type the prefix at the Swat prompt, then type Ctrl-D. To automatically complete a command name use the Escape key (if the prefix is unambiguous) or Ctrl-] to scroll through the list of possible command completions.



Swat Introduction

72

◆ Initialization Files

If there are certain Swat commands that always need to be executed when Swat is run, then they can be placed in an initialization file. (See Swat Display 3-16) An initialization file contains a list of commands that will be executed just before the first prompt in Swat.

The initialization file should be called SWAT.RC. Swat will look in the directory from which it was invoked for such a file. If it doesn't find one there, it will look for a file named SWAT.RC in a directory named in the HOME environment variable

3.6

Swat Display 3-16 An Initialization File

```
srcwin 15
regwin
save 500
patient-default mess1
run
```

This example shows a sample initialization file which sets up windows to display the source code and current register values, set the length of the save buffer to 500 lines, and continue running swat until the mess1 application has been loaded, at which point execution will automatically stop.



Swat Reference



4.1	Notation	75
4.2	Swat Reference	76

4





This chapter is intended to provide documentation for the majority of useful Swat commands. The general structure of the descriptions in this chapter will be as follows:

Usage:	Shows the command and its various arguments and subcommands (if any).
Examples:	Examples of the command as it could be used.
Synopsis:	Summary of the command and its functions and results.
Notes:	Details about the subcommands, arguments, and other command features.
See Also:	Other related commands.

4.1

4.1 Notation

The descriptions of the Swat commands will follow the following notational conventions:

- ◆ **command (alternative1 | alternative2 | ... | alternativeN)**
 () The parentheses enclose a set of alternatives separated by a vertical line. For example, the expression **quit (cont | leave)** means that either **quit cont** or **quit leave** can be used.
- ◆ **command [optional_argument]**
 [] The brackets enclose optional arguments to the command. For example, the command **alias [<word[<command>]>]** could have zero, one, or two arguments because the *<command>* and *<word>* arguments are optional. Another example would be the command **objwalk [<addr>]**, which may take zero arguments if it is meant to use the default address, and one argument if the user gives it a particular address to look at.
- ◆ **command <type_of_argument>**
 < > The angled brackets enclose the type of an argument rather than the actual string to be typed. For example, **<addr>** indicates an address expression and **<argument>** indicates some sort of argument, but **(addr | type)** means either the string **addr** or the string **type**.



◆ * +

An asterisk following any of the previous constructs indicates zero or more repetitions of the construct may be typed. An addition sign indicates one or more repetitions of the construct may be used. For example, **unalias word*** can be the **unalias** command by itself, or it can be followed by a list of words to be unaliased.

4.2

4.2 Swat Reference

■ **_print**

Usage: _print <expression>

Examples:

“_print ax-10”
 print ax less 10 decimal.

Synopsis: Print the value of an expression.

Notes: The difference between this command and the “print” command is a subtle one: if one of the arguments contains square-brackets, the Tcl interpreter will attempt to evaluate the text between the brackets as a command before _print is given the argument. If the text between the brackets is intended to be an array index, the interpreter will generate an error before the Swat expression evaluator has a chance to decide whether the text is a nested Tcl command or an array index.

For this reason, this function is intended primarily for use by Tcl procedures, not by users.

See Also: print, addr-parse.

■ **abort**

Usage: abort [<frame-number>]
 abort [<function>]

Examples:

“abort” abort executing the current frame.



“abort 3” abort executing up through the third frame.

“abort ObjMessage”
 abort executing up through first ObjMessage.

Synopsis: Abort code execution up through a given frame or routine. By “abort”, we mean “do not execute”. This can be quite dangerous, as semaphores may not be ungrabbed, blocks not unlocked, flags not cleared, etc., leaving the state of objects, and if executing system code, possibly the system itself in a bad state. This command should only be used when the only alternative is to detach (i.e. in a fatal error) as a way to possibly prolong the usefulness of the debugging session. 4.2

Notes:

- ◆ If no argument is given, code through the current frame is aborted.
- ◆ <frame num> are the numbers that appear at the left of the backtrace.

See Also: finish, backtrace, zip.

■ abortframe

Usage: abortframe <frame-token>

Examples:

“abortframe \$cur”
 Abort all code execution through the frame whose token is in \$cur.

Synopsis: Aborts code execution up through a particular stack frame. As no code is executed, the registers may be in a garbaged state.

Notes:

- ◆ The argument is a frame token, as returned by the “frame” command.
- ◆ No FULLSTOP event is dispatched when the machine actually aborts executing in the given frame. The caller must dispatch it itself, using the “event” command. For information about FULLSTOP events, see the **event** Tcl command.
- ◆ The command returns zero if the machine aborted executing in the given frame; non-zero if it was interrupted before that could happen.

■ addr-parse

Usage: addr-parse <addr> [<addr-only>]

Examples:

“addr-parse *ds:si”

Parse the address “*ds:si” into its handle, offset and data-type components. In this case, the data-type will be “nil”.

4.2

“addr-parse ILLEGAL_HANDLE 0”

Figures the value for the enumerated constant “ILLEGAL_HANDLE”. The handle for this non-address will be “value”.

Synopsis: This command parses the address expression into its components, returning a list {<handle> <offset> <type> } as its value.

Notes:

- ◆ This will generate an error if there’s an error parsing the <addr>
- ◆ <handle> is the token for the handle in which the address resides, or “nil” if the address is absolute. This token can be given to the “handle” command for further processing.
- ◆ <offset> is a decimal number and is the offset of the address within the block indicated by the <handle> token. If <handle> is “nil”, this can be a 32-bit linear address.
- ◆ <type> is a type token for the data at the given address, if any could be determined. For example the address “ds:bx” has no type, as it’s just a memory reference, but “ds:bx.VDE_extraData” will have whatever type the structure field “VDE_extraData” possesses. This token can be passed to the “type” or “value” commands for further processing.
- ◆ If the expression doesn’t refer to data that can be fetched from the patient (e.g. “foo*3”) <handle> will be returned as the string “value” instead of a normal handle token. <offset> is then a value-list for the resulting value, and <type> is the type description by means of which the value list can be interpreted.
- ◆ The optional <addr-only> argument is zero or non-zero to indicate the willingness or unwillingness, respectively, of the caller to receive a value list in return. If <addr-only> is absent or non-zero, any expression that can only be expressed as a value will generate an error. The single



exception to this is if the expression involves pointer arithmetic. For example “pself+1” normally would be returned as a value list for a far pointer, as the result cannot be fetched from the PC. When <addr-only> is absent or non-zero, “addr-parse” pretends the expression was “*(pself+1)”, allowing simple specification of an address by the user for those commands that just address memory.

- ◆ The <offset> element of the returned list is very useful when you want to allow the user to give you anything, be it a register or a number or an enumerated constant or whatever. You can pass the argument you were given to [index [addr-parse \$arg] 1] and end up with an appropriate decimal number. Be sure to pass <addr-only> as 0, however, or else you’ll generate an error.

4.2

See Also: value, handle, type.

■ addr-preprocess

Usage: addr-preprocess <addr> <seg-var> <off-var>

Examples:

“addr-preprocess \$args s o”

Parse the address expression in \$args, storing the segment portion in \$s and the offset portion in \$o in the current scope.

Synopsis: Preprocesses an address expression into a form that is easier to manipulate and faster to reparse.

Notes:

- ◆ <seg-var> is the name of a variable in the caller’s scope in which the segment of the address is stored. It should be treated as opaque, as it may or may not be numeric.
- ◆ <off-var> is the name of a variable in the caller’s scope in which the offset of the address is stored. This will always be numeric.
- ◆ Returns the 3-list returned by addr-parse, in case you have a use for the type token stored in the list.

See Also: addr-parse.

■ addr-with-obj-flag

Usage: addr-with-obj-flag

Swat Reference

80

Examples:

`"var addr [addr-with-obj-flag $addr]"`
If \$addr is "-i", returns the address of the current implied grab.

Synopsis:

This is a utility routine that can be used by any command that deals with objects where the user may reasonably want to operate on the leaf object of one of the hierarchies, or the windowed object under the mouse. It can be given one of a set of flags that indicate where to find the address of the object on which to operate.

4.2

Notes:

- ◆ Special values accepted for <address>:

Value Returns address expression for...

- a the current patient's application object
- i the current "implied grab": the windowed object over which the mouse is currently located
- f the leaf of the keyboard-focus hierarchy
- t the leaf of the target hierarchy
- m the leaf of the model hierarchy
- c the content for the view over which the mouse is currently located
- kg the leaf of the keyboard-grab hierarchy
- mg the leaf of the mouse-grab hierarchy

- ◆ If <address> is empty, this will return the contents of the local variable "oself" within the current frame, if it has one, or *ds:si
- ◆ If <address> isn't one of the above, this just returns <address>.

See Also:

impliedgrab, content, focusobj, targetobj, modelobj, keyboardobj, mouseobj.

■ alias

Usage: alias [<name> [<body>]]

Examples:



“alias p print”

Execute “print” when the user types the command “p”. Any arguments to “p” get passed to “print” in the order they were given.

“alias while {for {} \$1 {} \$2}”

Executes an appropriate “for” loop when the “while” command is executed with its two arguments: a test expression and a body of commands to execute.

“alias”

Prints all the defined aliases.

4.2

“alias while”

Prints what the “while” command is aliased to.

Synopsis:

This is a short-cut to allow you to make commands you commonly type easier to use, and to define simple new commands quickly.

Notes:

- ◆ If you give no arguments the current aliases are all displayed.
- ◆ If you give a single argument, the name of an existing alias, the command that will be executed when you use the alias is printed.
- ◆ The <body> string is usually in curly braces, as it usually involves whitespace and can contain newlines for the longer aliases.
- ◆ You can use the pseudo-variables \$1, \$2, etc. in the <body> to represent the 1st, 2nd, etc. argument given when the alias is invoked. They are pseudo-variables as the “var” command will not operate on them, nor are they available to any procedure invoked by the alias.
- ◆ You can also interpolate a range of the arguments using \$<start>--<end>. If you do not give an <end>, then the arguments from <start> to the last one will be interpolated.
- ◆ \$* will interpolate all the arguments.
- ◆ \$# will interpolate the actual number of arguments.
- ◆ If you do not use any of these pseudo-variables, all the arguments given to the alias will be appended to the <body>.
- ◆ Interpolation of the values for these pseudo-variables occurs regardless of braces in the <body>.

- ◆ It is an error to specify an argument number when there are fewer than that many arguments given to the alias.

See Also: unalias.

■ alignFields

Usage: var alignFields [(0 | 1)]

Examples:

4.2

“var alignFields 1”

Sets the “print” command to align the values for all the fields of a given structure.

Synopsis: Determines whether structure-field values follow immediately after the field name or if all values are indented to the same level. The “print” command and other display commands use this variable when formatting their output.

Notes:

- ◆ Having all values indented to the same level makes it easier for some people to locate a particular field in a structure. It is not without cost, however, in that Swat must determine the length of the longest field name before it can print anything.
- ◆ The default value for this variable is zero.

See Also: print.

■ antifreeze

Usage: antifreeze <patient>
antifreeze :<n>
antifreeze <patient>:<n>
antifreeze <id>

Examples:

“antifreeze term”

Promotes the application thread for “term” to be the “most-runnable”

“antifreeze :1”

Does likewise for thread #1 of the current patient



“antifreeze 16c0h”

Does likewise the thread whose handle is 16c0h

“antifreeze” Promotes the current thread to be the “most-runnable.”

See Also: freeze

■ antithaw

Usage: antithaw <patient>
antithaw :<n>
antithaw <patient>:<n>
antithaw <id>

4.2

Examples:

“antithaw term”

Allows the application thread for “term” to run normally.

“antithaw :1”

Allows thread #1 of the current patient to run normally.

“antithaw 16c0h”

Allow the thread whose handle is 16c0h to run normally.

See Also: thaw

■ appobj

Usage: appobj [<patient>]

Examples:

“pobj [appobj draw]”

prints the GenApplication object for draw.

“pobj [appobj]”

prints the GenApplication object for the current application
(equivalent to “pobj -a”).

Synopsis: Returns the address of the GenApplication object for the given patient, or the current one if you give no patient.

See Also: impliedgrab.

■ **apropos**

Usage: `apropos [<string>]`

Examples:

`“apropos vis”`
Find all commands related to vis

4.2

`“apropos text”`
Find all commands related to text

Synopsis: Search the list of commands and command help for all entries containing `<string>`. Lists each command and its synopsis.

Notes: `<string>` may actually be a pattern, as described in the help for the “string” command (under “string match”). It automatically has a leading and following `*` tacked onto it to cause it to match anywhere within a help string.

See Also: `help`.

■ **aset**

Usage: `aset <array-name> <index> <value>`

Examples:

`“aset foo $i $n”`
Sets the `$i`’th element (counting from zero) of the value stored in the variable `foo` to `$n`.

Synopsis: Allows you to treat a list stored in a variable as an array, setting arbitrary elements of that array to arbitrary values.

Notes:

- ◆ `<array-name>` is the name of the variable, not the value of the variable to be altered.
- ◆ This command returns nothing.
- ◆ The index must be within the bounds of the current value for the variable. If it is out of bounds, `aset` will generate an error.

See Also: `index`.



■ assoc

Usage: assoc <list> <key>

Examples:

“assoc \$classes GenPrimaryClass”

Examines the sublists of \$classes and returns the first one whose first element is the string GenPrimaryClass.

Synopsis: Searches an associative list to find an element with a particular key. The list is itself made up of lists, each of whose first element is a key.

4.2

Notes:

- ◆ A typical associative list is made of key/value pairs, like this:
{{<key> <value>} {<key> <value>} ...}
- ◆ If an element is found whose <key> matches the passed <key>, the entire element is returned as the result. If no <key> matches, nil is returned.

See Also: car, cdr, range, list, delassoc.

■ assign

Usage: assign <addr> <value>

Examples:

“assign ip ip+2”

Add 2 to the value of IP in the current frame.

“assign {word ds:si} 63h”

Store 63h in the word at ds:si

Synopsis: Performs an assignment to a patient variable or register (but not to an element of the value history). The first argument is the variable or register to be assigned and the second argument is the value to assign to it (which may be a regular address expression). If the first expression doesn't indicate a type, “word” is assumed. Only **byte**, **word** or **dword** types are supported.

Notes:

- ◆ When assigning to an sptr, the value assigned will be the segment of the block indicated by the <value>, unless <value> is an absolute address (or

Swat Reference

86

just a number), in which case the low 16 bits of the offset will be used instead.

- ◆ Similar behavior occurs when assigning to an `fp`tr, except if the `<value>` is an absolute address, in which case the linear address in the offset portion of the `<value>` will be decomposed into a segment and an offset.

See Also: `imem`, `value`

4.2

■ `att`

Usage: `att [<args>]`

Examples:

`"att"` attach Swat to GEOS.

Synopsis: Attach Swat to GEOS.

Notes: The `args` argument can be one of the following:

`-s` reboot GEOS with error checking, attach, and stop
`-sn` reboot GEOS without error checking, attach, and stop
`-f` restart GEOS with error checking and attach after a pause
`-r` restart GEOS with error checking and attach
`-rn` restart GEOS without error checking and attach

See Also: `detach`, `quit`.

■ `attach`

Usage: `attach [<boot>]`

Examples:

`"attach"` attach to the target PC

Synopsis: Attach swat to the target PC.

Notes:

- ◆ The `boot` argument is `"-b"` to bootstrap and `"+b"` to not. Normally, Swat will try to read symbolic information about all running geodes; bootstrapping specifies that Swat should only read symbolic information for these geodes when it must.



- ◆ If you give no <boot> argument, swat will use the most-recent one.
- ◆ By default, swat will locate the symbols for all geodes and threads active on the PC when it attaches.
- ◆ If any geode has changed since you detached from the PC, its symbols are re-read.

See Also: att, detach, quit.

■ autoload

4.2

Usage: autoload <function> <flags> <file> [<class> <docstring>]

Examples:

“autoload cycles 1 timing”

load the file “timing.tcl” when the cycles command is first executed. The user must type the command completely.

“autoload print 2 print”

load the file “print.tcl” when the print command is first executed. The user may abbreviate the command and the Tcl interpreter will not evaluate its arguments.

Synopsis: This command allows the first invocation of a command to automatically force the transparent reading of a file of Tcl commands.

Notes:

- ◆ autoload takes 3 or 5 arguments: the command, an integer with bit flags telling how the interpreter should invoke the command, the file that should be read to define the command (this may be absolute or on load-path) and an optional help class and string for the command.
- ◆ The help class and string need only be given if the file to be loaded isn't part of the system library (doesn't have its help strings extracted when Swat is built).
- ◆ The <flags> argument has the following bit-flags:

0	User must type the command's name exactly. The command will be defined by “defsubr” or “defdsbr” when <file> is loaded.
1	The interpreter will not evaluate arguments passed to the command. All arguments will be merged into a single string

and passed to the command as one argument. The command will use the special “noeval” argument when it is defined.

See Also: defsubr, defdsubr, defcommand, proc.

■ backtrace

Usage: backtrace [-r<reg>*][<frames to list>]

Examples:

4.2

“backtrace” print all the frames in the patient

“backtrace -rax”
print all the frames and the contents of AX in each one.

“where 5” print the last five frames

“w 5” print the last five frames

Synopsis: Print all the active stack frames for the patient.

Notes:

- ◆ The <frames to list> argument is the number of frames to print. If not specified, then all are printed.
- ◆ If a numeric argument is not passed to backtrace then it attempts to display method calls in the form:

```
MSG_NAME(cx, dx, bp) => className (^l####h:####h)
```

Here <cx>, <dx>, and <bp> are the values passed in these registers. <className> is the name of the class which handled the message. ^l####h:####h is the address of the object (block, chunk handle) handling the message.

- ◆ If a numeric argument is passed to backtrace then the attempt to decode the message is not done and the single line above expands into:

```
far ProcCallModuleRoutine(), geodesResource.asm:476  
near ObjCallMethodTable(), objectClass.asm:1224
```

This is generally less useful, but sometimes it's what you need.

See Also: up, down, func, where.



■ bindings

Usage: bindings
Synopsis: Shows all current key bindings

■ bind-key

Usage: bind-key <ascii_value> <function>

Examples:

“bind-key \321 scroll_srcwin_down”
Binds scroll-down key to the scroll_srcwin_down Tcl routine.

Synopsis: Binds an ASCII value to a function.

See Also: alias, unbind-key.

■ break This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ break-taken

Usage: break-taken [<flag>]

Examples:

“break-taken”
Returns 1 if the machine stopped because of a breakpoint.

“break-taken 0”
Specify that no breakpoint was actually taken to stop the machine.

Synopsis: Obscure. This is used to determine if the machine stopped because a breakpoint was hit and taken.

Notes: Setting the **break-taken** flag is a rather obscure operation. It is useful primarily in complex commands that single-step the machine until a particular address is reached, or a breakpoint is taken when a breakpoint must be used to skip over a procedure call, or condense multiple iterations of an instruction with a REP prefix into 1. For an example of this use, refer to the “cycles” command.

See Also: brk, irq.

■ **brk** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ **brkload**

Usage: brkload [<handle>]

Examples:

4.2

“brkload Interface”

Stop the machine when the Interface resource is loaded or swapped in.

“brkload bx” Stop the machine when the resource whose handle ID is in BX is loaded or swapped in.

“brkload” Stop watching for the previously-specified resource to be loaded.

Synopsis: Stop the machine when a particular resource is loaded into memory.

Notes:

- ◆ Only one brkload may be active at a time; registering a second one automatically unregisters the first.
- ◆ If you give no <handle> argument, the previously-set brkload will be unregistered.

See Also: handle.

■ **byteAsChar**

Usage: var byteAsChar [(0 | 1)]

Examples:

“var byteAsChar 1”

Print byte variables as characters.

Synopsis: Determines how unsigned character variables are printed: if set non-zero, they are displayed as characters, else they are treated as unsigned integers.

Notes:

- ◆ If *\$byteAsChar* is 0, *\$intFormat* is used.
- ◆ The default value for this variable is 0.



■ bytes

Usage: bytes [<address>] [<length>]

Examples:

“bytes” lists 16 bytes at DS:SI

“bytes ds:di 32”
lists 32 bytes at DS:SI

4.2

Synopsis: Examine memory as a dump of bytes and characters.

Notes:

- ◆ The <address> argument is the address to examine. If not specified, the address after the last examined memory location is used. If no address has been examined then DS:SI is used for the address.
- ◆ The <length> argument is the number of bytes to examine. It defaults to 16.
- ◆ Pressing <Return> after this command continues the list.
- ◆ Characters which are not typical ASCII values are displayed as a period.

See Also: words, dwords, imem, assign.

■ cache

This is a Tcl primitive. See page 306.

■ call

Usage: call <function> [<function args>]

Examples: “call MyFunc”
“call MyDraw ax 1 bx 1 cx 10h dx 10h”
“call FindArea box.bottom 5 box.right 5 push box”

Synopsis: Call a function in the current thread.

Notes:

- ◆ The <function> argument is the function to call. If it is a NEAR function, the thread must already be executing in the function's segment.

4.2

- ◆ The function arguments are in pairs <variable/register> <value>. These pairs are passed to the “assign” command. As a special case, if the variable is “push”, the value (a word) is pushed onto the stack and is popped when the call finishes (if it completes successfully).
- ◆ All current registers are preserved and restored when the call is complete. Variables are not.
- ◆ Once the call has completed, you are left in a sub-interpreter to examine the state of the machine. Type “break” to get back to the top level.
- ◆ If the machine stops for any other reason than the call’s completion, the saved register state is discarded and you are left wherever the machine stopped. You will not be able to get a stack trace above the called function, but if the call eventually completes, and no registers have actually been modified, things will get back on track.
- ◆ You may not call a function from a thread that has retreated into the kernel. This function also will not allow you to call **ThreadExit()**. Use the “exit” function to do that.

See Also: assign, call-patient, patch.

■ call-patient

Usage: call-patient <function> ((<reg> | push) <value>)*

Examples:

“call-patient MemLock bx \$h”
Locks down the block whose handle ID is in \$h.

Synopsis: This is a utility routine, not intended for use from the command line, that will call a routine in the PC after setting registers to or pushing certain values.

Notes:

- ◆ Returns non-zero if the call completed successfully.
- ◆ If the call is successful, the registers reflect the state of the machine upon return from the called routine. The previous machine state is preserved and can be retrieved, by invoking restore-state, or thrown away, by invoking discard-state. The caller *must* invoke one of these to clean up.
- ◆ Arguments after <function> are as for “call”.



- ◆ If the called routine is in movable memory, this will lock the containing block down before issuing the call, as you'd expect.
- ◆ Calling anything that makes message calls while on the geos:0 thread is a hazardous undertaking at best.

See Also: call.

■ car

Usage: car <list>

4.2

Examples:

“car \$args” Returns the first element of \$args.

Synopsis: Returns the first element of a list.

Notes: This is a lisp-ism for those most comfortable with that language. It can be more-efficiently implemented by saying [index <list> 0]

See Also: cdr.

■ **case** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ **catch** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ **cbrk** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ cdr

Usage: cdr <list>

Examples:

“cdr \$args” Returns the remaining arguments yet to be processed.

Synopsis: Returns all but the first element of a list.

Notes: This is a lisp-ism for those most comfortable with that language. It can be more-efficiently implemented by saying [range <list> 1 end]

See Also: car.

■ classes

Usage: classes [<patient>]

Examples:

“classes ” Print list of classes in current patient.

“classes myapp”
 Print list of classes in myapp patient.

Synopsis: Prints list of classes defined by the given patient.

Notes: Remember that “brk” will take address arguments of the form
 <class>::<message>, so you can use this function and set a breakpoint using
 “brk MyTextClass::MSG_MY_TEXT_MESSAGE”. If you need a breakpoint
 that’s limited to one object, use objbrk instead.

4.2

■ clrcc

Usage: clrcc <flag> [<value>]

Examples:

“clrcc c” clear the carry flag

Synopsis: Clear a flag in the target computer.

Notes: The first argument is the first letter of the flag to clear. The following is a list
 of the flags:

t	trap
i	interrupt enable
d	direction
o	overflow
s	sign
z	zero
a	auxiliary carry
p	parity
c	carry

See Also: setcc, compcc, getcc.

■ columns

Usage: columns

Examples:

“columns”

Return the number of columns on the screen.

Synopsis: Retrieves the width of the screen, if known, to allow various commands (most notably “print”) to size their output accordingly.

■ compcc

Usage: compcc <flag>

Examples:

4.2

“compcc c” complement the carry flag

Synopsis: Complement a flag in the target computer.

Notes: The first argument is the first letter of the flag to complement. The following is a list of the flags:

t	trap
i	interrupt enable
d	direction
o	overflow
s	sign
z	zero
a	auxiliary carry
p	parity
c	carry

This command is handy to insert in a patch to flip a flag bit.

See Also: setcc, clrc.

■ completion

Usage: completion <list-of-names>

Examples:

“completion {martial marital}”
Returns “mar,” the common prefix.

Synopsis: Figures the common prefix from a set of strings. Used for the various forms of completion supported by top-level-read.

See Also: top-level-read.

■ **concat** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ **condenseSmall**

Usage: var condenseSmall [(0 | 1)]

Examples:

“var condenseSpecial 0”

Force even small structures to be printed one field per line.

4.2

Synopsis: Controls whether “print” attempts to condense the output by printing small (< 4 bytes) structures (which are usually records in assembly language) as a list of <name> = <int>, where <name> is the field name and <int> is a signed integer.

Notes: The default value of this variable is one.

See Also: print, condenseSpecial.

■ **condenseSpecial**

Usage: var condenseSpecial [(0 | 1)]

Examples:

“var condenseSpecial 0”

Turns off the special formatting of various types of structures by “print”.

Synopsis: Controls the formatting of certain structures in more-intuitive ways than the bare structure fields.

Notes:

- ◆ The default value of this variable is 1.
- ◆ The current list of structures treated specially are: **Semaphore**, **Rectangle**, Output Descriptor, **TMatrix**, **BBFixed**, **WBFixed**, **WWFixed**, **DWFixed**, **WDFixed**, **DDFixed**, **FileDate**, **FileTime**, **FloatNum**, **SpecWinSizeSpec**.

See Also: print, condenseSmall.

■ **cont**

Usage: cont



Examples:

“cont” continue execution
 “c” continue execution

Synopsis: Continue GEOS.

Notes:

- ◆ If the global variable *waitForPatient* is non-zero, this command waits for the machine to stop again before it returns.

4.2

See Also: go, istep, step, next, detach, quit.

■ content

Usage: content

Examples:

“vistree [content]”
 print the visual tree of the content of the view under the mouse.

Synopsis: Print the address of the content under the view with the current implied grab.

Notes:

- ◆ This command is normally used with vistree to get the visual tree of a content by placing the mouse on the content's view window and issuing the command in the example.
- ◆ If the pointer is not over a GenView object, this is the same as the “impliedgrab” command.

See Also: systemobj, gentree, impliedgrab.

■ continue This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ continue-patient

Usage: continue-patient

Examples:

Swat Reference

98

“continue-patient”

Allow the target machine to continue executing GEOS.

Synopsis: Tell the Swat stub to let the target machine continue where it left off.

Notes: This command does not wait for the machine to stop again before it returns; once the machine is running, you’re free to do whatever you want, whether it’s calling “wait” or examining memory periodically.

4.2

See Also: step-patient.

■ cup

Usage: cup <class>
cup <object>
cup <flags>

Examples:

“cup ui::GenDocumentControlClass”
Print class hierarchy of named class

“cup ^l2850h:0034h”
Print class hierarchy of object

“cup -f” Print class hierarchy of focus object

“cup -p” Print class hierarchy of process

Synopsis: Walks up the class hierarchy, starting at a given class, printing each class encountered. May be passed an object, in which case the class of the object will be used as a starting place.

■ current-level

Usage: current-level

Examples:

“var l [current-level]”
Store the current interpreter nesting level in \$l.

Synopsis: Returns the number of invocations of “top-level” (i.e. the main command input loop) currently active.

Notes:



- ◆ This is currently used only to modify the command prompt to indicate the current nesting level.
- ◆ The top-most command loop is level one.

See Also: prompt, top-level.

■ current-registers

Usage: current-registers

4.2

Examples:

“current-registers”

Returns a list of the current registers for the current thread.

Synopsis: Returns all the registers for the current thread as a list of decimal numbers.

Notes:

- ◆ The mapping from element number to register name is contained in the global variable “regnums”, which is an assoc-list whose elements contain the name of the register, then the element number.
- ◆ For your own consumption, the list is ordered ax, cx, dx, bx, sp, bp, si, di, es, cs, ss, ds, ip, flags. You should use the “regnums” variable when programming, however, as this may change at some point (e.g. to accommodate the additional registers in the 386).

■ cvtrecord

Usage: cvtrecord <type> <number>

Examples:

“cvtrecord [symbol find type HeapFlags] 36”

Return a value list for the number 36 cast to a **HeapFlags** record.

Synopsis: Creates a value list for a record from a number, for use in printing out the number as a particular record using **fmtval**.

Notes:

- ◆ <type> is a type token for a record (or a structure made up exclusively of bitfields).

- ◆ <number> must be an actual number suitable for the “expr” command. It cannot be a register or variable or some such. Use “getvalue” to obtain an integer from such an expression.
- ◆ Returns a value list suitable for “value store” or for “fmtval”.

See Also: value, fmtval, expr, getValue.

■ cycles

4.2 Synopsis: Count instruction cycles from now until the given address is reached. Prints out each instruction as it is executed, along with the cycles it took. If no address is given, executes until a breakpoint is hit. Takes the following (optional) flags:

- r Print routines called, the total cycles for each routine, and a running total, not the cycles for each instruction.
- i Same as -r, but indents to show calling level. Not recommended for counting cycles over deeply nested routines.
- I Same as -i, except uses (#) to indicate call level
- f Stop counting when this routine finishes
- n Does not whine about interrupts being off
- x <routine>
Step over calls to <routine>
- x <routine>=<val>
Step over calls to <routine> and assume that the call takes <val> cycles for timing purposes

■ dcache

Usage: dcache bsize <blockSize>
dcache length <numBlocks>
dcache stats
dcache params
dcache (on | off)

Examples:

“dcache bsize 16”
Set the number of bytes fetched at a time to 16.



“dcache length 1024”

Allow 1024 blocks of the current block size to be in the cache at a time.

“dcache off” Disables the Swat data cache.

Synopsis: Controls the cache Swat uses to hold data read from the target machine while the machine is stopped.

Notes:

4.2

- ◆ Data written while the machine is stopped actually get written to the cache, not the PC, and the modified blocks are written when the machine is continued.
- ◆ The default cache block size is 32 bytes, with a default cache length of 64 blocks.
- ◆ It is a very rare thing to have to turn the data cache off. You might need to do this while examining the changing registers of a memory-mapped I/O device.
- ◆ The `<blockSize>` must be a power of 2 and no more than 128.
- ◆ Changing the block size causes all cached blocks to be flushed (any modified cache blocks are written to the PC).
- ◆ Changing the cache length will only flush blocks if there are more blocks currently in the cache than are allowed by the new length.
- ◆ The “dcache stats” command prints statistics giving some indication of the efficacy of the data cache. It does not return anything.
- ◆ The “dcache params” command returns a list `{<blockSize> <numBlocks>}` giving the current parameters of the data cache. There are some operations where you might want to adjust the size of the cache either up or down, but need to reset the parameters when the operation completes. This is what you need to do this.

See Also: cache.

■ dcall

Usage: dcall [<args>]

Examples:

“dcall Dispatch”

Display when the routine Dispatch is called

“dcall none”

stop displaying all routines

Synopsis: Display calls to a routine.

Notes:

4.2

- ◆ The <args> argument normally is the name of the routine to monitor. Whenever a call is made to the routine its name is displayed.
- ◆ If ‘none’ or no argument is passed, then all the routines will stop displaying.
- ◆ Dcall uses breakpoints to display routine names. By looking at the list of breakpoints you can see which routines display their names and you can stop them individually by disabling or deleting their breakpoints.

See Also: showcalls, mwatch.

■ debug

Usage: debug <proc-name>*

Examples:

“debug” Enter the Tcl debugger immediately.

“debug fooproc”

Enter the Tcl debuffer when the interpreter is about to execute the command “fooproc”.

Synopsis: This command is used when debugging Tcl commands. It sets a breakpoint at the start of any Tcl command. Also serves as a breakpoint in the middle of a Tcl procedure, if executed with no argument.

Notes:

- ◆ The breakpoint for <proc-name> can be removed using the “undebug” command.
- ◆ <proc-name> need not be a Tcl procedure. Setting a breakpoint on a built-in command is not for the faint-of-heart, however, as there are some commands used by the Tcl debugger itself. Setting a breakpoint on such a command will cause instant death.



See Also: `undebug`.

■ **debugger**

Usage: `var debugger [<command-name>]`

Synopsis: Name of the command when things go wrong. The function is passed two arguments: a condition and the current result string from the interpreter. The condition is “enter” if entering a command whose debug flag is set, “exit” if returning from a frame whose debug flag is set, “error” if an error occurred and the “debugOnError” variable is non-zero, “quit” if quit (^) is typed and the “debugOnReset” variable is non-zero, or “other” for some other cause (e.g. “debug” being invoked from within a function).

4.2

■ **debugOnError**

Usage: `var debugOnError [(0 | 1)]`

Examples:

`“var debugOnError 1”`
Turn on debugging when there’s a Tcl error.

Synopsis: Enter debug mode when Swat encounters a Tcl error.

Notes:

- ◆ The 0 | 1 simply is a false | true to stop and debug upon encountering an error in a Tcl command.
- ◆ If an error is caught with the catch command, Swat will not enter debug mode.

See Also: `debugger`.

■ **defcmd**

Usage: `defcmd <name> <args> <help-class> <help-string> <body>`

Examples: Look at almost any .tcl file in the system library for an example; a complete example set would be too large to give here.

Synopsis: This creates a new Tcl procedure with on-line help whose name the user may abbreviate when invoking.

Notes:

4.2

- ◆ `<help-class>` is a Tcl list of places in which to store the `<help-string>`, with the levels in the help tree separated by periods. The leaf node for each path is added by this command and is `<name>`, so a command “foo” with the `<help-class>` “prog.tcl” would have its `<help-string>` stored as “prog.tcl.foo.”
- ◆ Because the name you choose for a procedure defined in this manner can have an impact on the unique abbreviation for another command, you should use this sparingly.

See Also: defcommand, proc, help.

■ defcommand

Usage: defcommand `<name>` `<args>` `<help-class>` `<help-string>` `<body>`

Examples: Look at Swat Display 5-3, Swat Display 5-4, or almost any .tcl file in the system library for an example; a complete example set would be too large to give here.

Synopsis: This creates a new Tcl procedure with on-line help whose name must be given exactly when the user wishes to invoke it.

Notes: `<help-class>` is a Tcl list of places in which to store the `<help-string>`, with the levels in the help tree separated by periods. The leaf node for each path is added by this command and is `<name>`, so a command “foo” with the `<help-class>` “prog.tcl” would have its `<help-string>` stored as “prog.tcl.foo.”

See Also: defcmd, proc, help.

■ defhelp

Usage: defhelp `<topic>` `<help-class>` `<help-string>`

Examples:

“defhelp breakpoint top {Commands relating to the setting of breakpoints}”
Sets the help for “breakpoint” in the “top” category to the given string.

Synopsis: This is used to define the help string for an internal node of the help tree (a node that is used in the path for some other real topic, such as a command or a variable).



Notes:

- ◆ This cannot override a string that resides in the **/pcgeos/tcl/doc** file.
- ◆ You only really need this if you have defined your own help-topic category.
- ◆ `<help-class>` is a Tcl list of places in which to store the `<help-string>`, with the levels in the help tree separated by periods. The leaf node for each path is added by this command and is `<name>`, so a command “foo” with the `<help-class>` “prog.tcl” would have its `<help-string>` stored as “prog.tcl.foo.”

4.2

See Also: help.

■ **defsubr** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ **defvar**

Usage: defvar `<name>` `<value>` [`<help-class>` `<help-string>`]**Examples:**`“defvar printRegions 0”`

Define “printRegions” as a global variable and give it the value zero, if it didn’t have a value already.

Synopsis: This command is used in .tcl files to define a global variable and give it an initial value, should the variable not have been defined before.**Notes:**

- ◆ If the variable is one the user may want to change, give it on-line help using the `<help-class>` and `<help-string>` arguments.
- ◆ `<help-class>` is a Tcl list of places in which to store the `<help-string>`, with the levels in the help tree separated by periods. The leaf node for each path is added by this command and is `<name>`, so a command “foo” with the `<help-class>` “variable.output” would have its `<help-string>` stored as “variable.output.foo.”

See Also: var, help.

■ **delassoc**

Usage: delassoc `<list>` `<key>` [`<foundvar>` [`<elvar>`]]**Examples:**

“delassoc \$val murphy”

Returns \$val without the sublist whose first element is the string “murphy.”

Synopsis: Deletes an entry from an associative list.

Notes:

4.2

◆ <foundvar>, if given, is the name of a variable in the caller’s scope that is to be set non-zero if an element in <list> was found whose <key> matched the given one. If no such element was found (and therefore deleted), the variable is set zero.

◆ <elvar>, if given, is the name of a variable in the caller’s scope that receives the element that was deleted from the list. If no element was deleted, the variable remains untouched.

See Also: assoc.

■ detach

Usage: detach [<options>]

Examples:

“detach cont”

continue GEOS and quit swat

Synopsis: Detach swat from the PC.

Notes:

◆ The <option> argument may be one of the following: *continue*: continue GEOS and detach swat; *leave*: keep GEOS stopped and detach swat. Anything else causes swat to just detach.

See Also: attach, quit.

■ dirs

Usage: dirs

Synopsis: Prints the directory stack for the current thread.

See Also: pwd, stdpaths



■ discard-state

Usage: discard-state

Examples:

“discard-state”

Throw away the values for all the thread's registers as saved by the most recent call to **save-state**.

Synopsis: Throw away the state saved by the most-recent **save-state** command.

4.2

Notes: This is usually only used in response to an error that makes it pointless to return to the point where the **save-state** was performed.

See Also: save-state, restore-state.

■ diskwalk

Usage: diskwalk <drive>

Examples:

“diskwalk F”

Prints the disks registered in drive F.

“diskwalk” Prints all the disks registered with the system.

Synopsis: Prints out the information on registered disks.

Notes: The Flags column is a string of single-character flags with the following meanings:

w The disk is writable.

V The disk is always valid, i.e. it's not removable.

S The disk is stale. This is set if the drive for the disk has been deleted.

u The disk is unnamed, so the system has made up a name for it.

See Also: drivewalk, fsdwalk.

■ display

Usage: display <lines> <command>
 display list
 display del <num>

Examples:

4.2 “display list”
 list all the commands displayed

 “display 1 {piv Vis VCNI_viewHeight}”
 always display the view height

 “display del 2”
 delete the second display command

Synopsis: Manipulate the display at the bottom of Swat’s screen.

Notes:

- ◆ If you give a numeric <lines> argument, the next argument, <command>, is a standard Tcl command to execute each time the machine halts. The output of the command is directed to a window <lines> lines high, usually located at the bottom of the screen.
- ◆ You can list all the active displays by giving “list” instead of a number as the first argument.
- ◆ If the first argument is “del”, you can give the number of a display to delete as the <num> argument. <num> comes either from the value this command returned when the display was created, or from the list of active displays shown by typing “display list”.

See Also: wtop, wcreate.

■ doc

Usage: doc [<keyword>]

Examples:

 “doc MSG_VIS_OPEN”
 Brings up technical documentation for MSG_VIS_OPEN.

Synopsis: Finds technical documentation for <keyword>. If it finds multiple entries for the keyword in the documentation, hit <Return> or use **doc-next** and



doc-previous to see the additional entries. The documentation retrieved is in ASCII form—figures will be missing, but the complete text appears.

See Also: doc-next, doc-previous.

■ doc-next

Usage: doc-next

Examples:

“doc MSG_VIS_OPEN”
Brings up technical documentation for MSG_VIS_OPEN.

“doc-next”
Brings up more technical documentation if available.

Synopsis: Finds additional technical documentation for <keyword>.

See Also: doc, doc-previous.

■ doc-previous

Usage: doc-next

Examples:

“doc MSG_VIS_OPEN”
Brings up technical documentation for MSG_VIS_OPEN.

“doc-next” Brings up more technical documentation if available.

“doc-previous”
Brings back previous (in this case, the first) entry.

Synopsis: Finds additional technical documentation for <keyword>.

See Also: doc, doc-next.

■ dosMem

Usage: dosMem

Examples:

“dosMem”

Synopsis: Traverse DOS’ chain of memory blocks, providing information about each.

■ down

Usage: down [<frame offset>]

Examples:

“down” move the frame one frame down the stack

“down 4” move the frame four frames down the stack

4.2 **Synopsis:** Move the frame down the stack.

Notes:

- ◆ The frame offset argument is the number of frames to move down the stack. If no argument is given then the current frame is moved down one frame.
- ◆ This command may be repeated by pressing <Return>.

See Also: backtrace, up.

■ drivewalk

Usage: drivewalk

Examples:

“drivewalk”
Prints the table of drives known to the system.

Synopsis: Prints out all disk drives known to the system, along with their current status.

Notes:

- ◆ The Flags column is a string of single character flags with the following meanings:
 - L The drive is accessible to the local machine only, i.e. it's not visible over a network.
 - R The drive is read-only.
 - F Disks may be formatted in the drive.
 - A The drive is actually an alias for a path on another drive.



- B The drive is busy, performing some extended operation, such as formatting or copying a disk.
- r The drive uses disks that may be removed by the user.
- n The drive is accessed over the network.

◆ The Locks column can reflect one of three states:

- none The drive isn't being accessed by any thread.
- Excl The drive is locked for exclusive access by a single thread.
- <num> The drive is locked for shared access for a particular disk, whose handle is the number. This is followed by the volume name of the disk, in square brackets.

4.2

See Also: diskwalk, fsdwalk.

■ dumpstack

Usage: dumpstack [<address>] [<length>]

Examples:

"dumpstack"
dump the stack at SS:SP

"ds ds:si 10"
dump ten words starting at DS:SI

Synopsis: Dump the stack and perform some simple interpretation upon it.

Notes:

- ◆ The <address> argument is the address of the list of words to dump. This defaults to SS:SP.
- ◆ The <length> argument is the number of words to dump. This defaults to 50.
- ◆ This dumps the stack and tries to make symbolic sense of the values, in terms of handles, segments, and routines.
- ◆ After doing a dumpstack, if you just hit return without entering a new command, by default you will see a continuation of the dumpstack.

See Also: backtrace.

■ dwordIsPtr

Usage: var dwordIsPtr [(0 | 1)]

Examples:

“var dwordIsPtr 1”

Tells “print” to print all double-word variables as if they were far pointers (segment:offset).

4.2

Synopsis: Controls whether dword (a.k.a. long) variables are printed as 32-bit unsigned integers or untyped far pointers.

Notes:

- ◆ For debugging C code, a value of 0 is more appropriate, while 1 is best for debugging assembly language.
- ◆ The default value for this variable is 1.

See Also: intFormat, print.

■ dwords

Usage: dwords [<address>] [<length>]

Examples:

“dwords” lists 4 double words at DS:SI

“dwords ds:di 8”
lists 8 double words at DS:DI

Synopsis: Examine memory as a dump of double words (32 bit hex numbers).

Notes:

- ◆ The <address> argument is the address to examine. If not specified, the address after the last examined memory location is used. If no address has been examined then DS:SI is used for the address.
- ◆ The <length> argument is the number of dwords to examine. It defaults to 4.
- ◆ Pressing <Return> after this command continues the list.

See Also: bytes, words, imem, assign.



■ **ec****Usage:** ec [<args>]**Examples:**

“ec” list the error checking turned on

“ec +vm” add vmem file structure checking

“ec all” turn on all error checking (slow)

“ec save none” save the current error checking and then use none

“ec restore” use the saved error checking flags

4.2

Synopsis: Get or set the error checking level active in the kernel.**Notes:** The following arguments may occur in any combination:

<***flag***> turn on <flag>

+<***flag***> turn on <flag>

-<***flag***> turn off <flag>

all turn on all error checking flags

ALL turn on all error checking flags

none turn off all error checking flags

sum <***handle***> turn on checksum checking for the memory block with the given handle (“ec sum bx”). The current contents of the block will be summed and that sum regenerated and checked for changes at strategic points in the system (e.g. when a call between modules occurs).

-sum turn off checksum checking

save save the current error checking

restore restore the saved error checking flags

where <flag> may be one of the following:

analVM perform over-aggressive checking of vmem files

	graphics	graphics checking
	heapFree	heap free block checking
	lmemFree	lmem free area checking
	lmemInternal	internal lmem error checking
4.2	lmemObject	lmem object checking
	normal	normal error checking
	region	region checking segment extensive
	segment	register checking
	lmemMove	force lmem blocks to move whenever possible
	unlockMove	force unlocked blocks to move whenever possible
	vm	vmem file structure checking
	vmemDiscard	force vmem blocks to be discarded if possible
	◆ If there isn't an argument, 'ec' reports the current error checking flags.	
	◆ Each time GEOS is run the ec flags are cleared. The saved flags are preserved between sessions. The ec flags may be saved and then restored after restarting GEOS so that the flag settings are not lost when restarting GEOS.	
	See Also:	why.

■ echo

Usage: echo [-n] <string>+

Examples:

"echo -n yes?"
Prints "yes?" without a newline.

"echo hi mom"
Prints "hi mom" followed by a newline.



Synopsis: Prints its arguments, separated by spaces.

Notes: If the first argument is “-n”, no newline is printed after the arguments.

See Also: flush-output

■ elist

Usage: elist [<patient>]

Examples:

4.2

“elist” list the events for the current thread and patient

“elist ui” list the events for the last thread of the ui patient

“elist :1” list the events for the first thread of the current patient

“elist geos:2” list the events for the second thread of the GEOS patient

Synopsis: Display all events pending for a patient.

Notes: The <patient> argument is of the form ‘patient:thread’. Each part of the patient name is optional, and if nothing is specified then the current patient is listed.

See Also: showcalls.

■ ensure-swat-attached

Usage: ensure-swat-attached

Examples:

“ensure-swat-attached”
Stop if Swat isn’t attached to GEOS.

Synopsis: If Swat is not attached to GEOS, display an error and stop a command.

Notes: Use this command at the start of any other command that accesses the target PC. Doing so protects the user from the numerous warnings that can result from an attempt to read memory when not attached.

■ eqfind

Usage: eqfind [-p]

Examples:

“eqfind” list all event queues in the system.

“eqfind -p” list and print all event queues in the system.

Synopsis: Display all event queues in the system.

See Also: elist, eqlist, erfind.

4.2 ■ **eqlist**

Usage: eqlist <queue handle> <name>

Examples:

“eqlist 8320 geos:2”
show the event list for geos:2

Synopsis: Display all events in a queue.

Notes:

- ◆ The queue handle argument is the handle to a queue.
- ◆ The name argument is the name of the queue.

See Also: elist.

■ **erfind**

Usage: erfind [-p]

Examples:

“erfind” list all recorded event handles in the system.

“erfind -p” list and print all recorded event handles in the system.

Synopsis: Display all record event handles in the system. These are events that have been recorded but not necessarily sent anywhere, so they will not appear in the queue of any thread.

See Also: elist, eqlist, eqfind, pevent.



- **error** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.
- **eval** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.
- **event** This is a Tcl primitive data structure. See “Tool Command Language,” Chapter 5.

- **exec** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ **exit**

4.2

Usage: exit <patient>

Examples:

“exit faxmon”

Causes the faxmon application to exit.

Synopsis: Sends messages required to make an application quit.

Notes: This command does nothing when you’re stopped at **FatalError**, as it will wait until the machine is idle before attempting to send MSG_META_QUIT; continuing from **FatalError** will cause the system to exit.

See Also: run.

■ **exit-thread**

Usage: exit-thread [<exit-code>]

Examples:

“exit-thread”

Exit the current thread, returning zero to its parent.

“exit-thread 1”

Exit the current thread, returning one to its parent.

Synopsis: Exit the current thread.

Notes:

- ◆ The exit code argument is the status to return to the current thread’s parent, which defaults to zero.
- ◆ Do not invoke this function for an event-driven thread; send it a MSG_META_DETACH instead.

See Also: quit.

■ explain

Usage: explain

Examples: “explain”

Synopsis: Print a more detailed description of why the system crashed, if possible.

4.2 Notes:

- ◆ This must be run from within the frame of the **FatalError()** function. Sometimes GEOS is not quite there. In this case, step an instruction or two and then try the “why” command again.
- ◆ This simply looks up the enumerated constant for the error code in the AX register in the **FatalErrors** enumerated type defined by the geode that called **FatalError()**. For example, if a function in the kernel called **FatalError()**, AX would be looked up in `geos::FatalErrors`, while if a function in your application called **FatalError()**, this function would look it up in the `FatalErrors` type defined by your application. Each assembly application defines this enumerated type by virtue of having included **ec.def**.
- ◆ This command also relies on programmers having explained their `FatalErrors` when defining them.
- ◆ For certain fatal errors, additional information is provided by invoking the command `<patient>::<error code name>`, if it exists.

■ explode

Usage: explode <string> [<sep-set>]

Examples:

“explode \$args”

Breaks the string stored in the variable “args” into a list of its individual letters.

“explode \$file /”

Breaks the string stored in the variable “file” into a list of its components, using “/” as the boundary between components when performing the split.



Synopsis: Breaks a string into a list of its component letters, allowing them to be handled quickly via a foreach loop, or the map or mapconcat commands.

Notes: This is especially useful for parsing command switches.

See Also: foreach, index, range.

■ **expr** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ **fatalerr_auto_explain**

4.2

Usage: var fatalerr_auto_explain [(0 | 1)]

Examples:

```
“var fatalerr_auto_explain 0”
    Turn off automatic generation of the explanation for any
    fatal-error hit.
```

Synopsis: Determines if the “why” command will automatically provide you with an explanation of any fatal error you encounter. If non-zero, they will be provided whenever **FatalError** is hit.

Notes:

- ◆ Explanations are loaded from <patient>.fei files stored in the system Tcl library directory when an error in <patient> is encountered.
- ◆ You can also obtain an explanation of an error via the “explain” command.

■ **fetch-optr**

Usage: fetch-optr <handle> <offset>

Examples:

```
“fetch-optr $h $o.GI_comp.CP_firstChild”
    Fetch the optr from the GI_comp.CP_firstChild field of the
    object at ^h$h:$o.
```

Synopsis: Extracts an optr from memory, coping with the data in the block that holds the optr not having been relocated yet.

Notes:

4.2

- ◆ <offset> is an actual offset, not a chunk handle, while <handle> is a handle ID, not a handle token.
- ◆ Returns a two-list {<handle> <chunk>}, where <handle> is the handle ID from the optr, and <chunk> is the chunk handle (low word) from the optr.
- ◆ We decide whether to relocate the optr ourselves based on the LMF_RELOCATED bit in the *LMBH_flags* field of the block's header. There are times, e.g. during the call to MSG_META_RELOCATE for an object, when this bit doesn't accurately reflect the state of the class pointer and we will return an error when we should not.

■ fhandle

Usage: fhandle <handle id>

Examples: "fhandle 3290h"

Synopsis: Print out a file handle.

Notes:

- ◆ The <handle id> argument is the handle number. File handles are listed in the first column of the 'fwalk' command.

See Also: fwalk.

■ field

Usage: field <list> <field name>

Examples:

"field [value fetch ds:si MyBox] topLeft"
return the offset of the topLeft field in MyBox

Synopsis: Return the value for the field's offset in the structure.

Notes:

- ◆ The <list> argument is a structure-value list from the "value" command.
- ◆ The <field name> argument is the field in the structure.

See Also: value, pobject, piv.



■ fieldwin**Usage:** fieldwin**Synopsis:** Print the address of the target machine's current top-most field window.

■ file This is a Tcl primitive. See "Tool Command Language," Chapter 5.

■ find**Usage:** find [-ir] <string> [<filename>]

4.2

Examples:`"find FileRead"`

Find next occurrence of string "FileRead" in currently viewed file

`"find FI_foo poof.goc"`

find first occurrence of string "FI_foo" in file poof.goc.

`"find -ir myobject"`

case-insensitive reverse search for most recent occurrence of string "myobject" in currently viewed file

Synopsis: Finds a string in a file and brings the line with that string to the middle of Swat's source window.**Notes:**

- ◆ If no file argument is specified, find will find the next instance of the string in the already viewed file starting from the current file position.
- ◆ There must already be a source window displayed for find to work.
- ◆ Possible options to find are:
 - r reverse search
 - i case insensitive search

■ find-opcode**Usage:** find-opcode <addr> <byte>+**Synopsis:** Locates the mnemonic for an opcode and decodes it. Accepts the address from which the opcode bytes were fetched, and one or more opcode bytes as arguments. Returns a list of data from the opcode descriptor:

{name length branch-type args modrm bRead bWritten inst}

length is the length of the instruction.

branch-type is one of:

1	none (flow passes to next instruction)
j	absolute jump
b	pc-relative jump (branch)
r	near return
R	far return
i	interrupt return
I	interrupt instruction

4.2

Any argument descriptor that doesn't match is to be taken as a literal. E.g. AX as a descriptor means AX is that operand.

modrm is the modrm byte for the opcode.

bRead is the number of bytes that may be read by the instruction, if one of its operands is in memory.

bWritten is the number of bytes that may be written by the instruction, if one of its operands is in memory.

inst is the decoded form of the instruction. If not enough bytes were given to decode the instruction, *inst* is returned as empty.

■ finish

Usage: finish [<frame num>]

Examples:

“finish” finish executing the current frame

“finish 3” finish executing up to the third frame

Synopsis: Finish the execution of a frame.

Notes:



- ◆ The <frame num> argument is the number of the frame to finish. If none is specified then the current frame is finished up. The number to use is the number which appears in a backtrace.
- ◆ The machine continues to run until the frame above is reached.

See Also: backtrace.

■ finishframe

Usage: finishframe [<frame-token>]

4.2

Examples:

“finishframe \$cur”

Run the machine to continue until it has returned from a particular stack frame.

Synopsis: Allows the machine to continue until it has returned from a particular stack frame.

Notes:

- ◆ No FULLSTOP event is dispatched when the machine actually finishes executing in the given frame. The caller must dispatch it itself, using the “event” command.
- ◆ The command returns zero if the machine finished executing in the given frame; non-zero if it was interrupted before that could happen.
- ◆ The argument is a frame token, as returned by the “frame” command.

See Also: event, frame, finish.

■ flagwin

Usage: flagwin [<on> | off]

Synopsis: Turns on or off a window providing a continuous display of the machine flags (e.g. zero, carry).

See Also: pflags.

■ flowobj

Usage: flowobj

Examples:

`"pobject [flowobj]"`
print out the flow object.

Synopsis: Prints out address of the uiFlowObj, which is the object which grabs the mouse.

Notes: This command is normally used with **pobject** to print out the object.

■ flush-output

4.2

Usage: flush-output

Examples:

`"flush-output"`
Forces pending output to be displayed.

Synopsis: Flushes any pending output (e.g. waiting for a newline) to the screen.

See Also: echo.

■ fmtree

Usage: fmtree <handle-id> <chunk>

Examples:

`"fmtree 3160h o"`
Prints a description of the object whose address is ^l3160h:0 (likely a thread/process).

Synopsis: Takes a global and a local handle and prints a description of the object described by that optr.

Notes:

- ◆ If the global handle is a thread or a process, the thread's name (process thread for a process handle) and the chunk handle (as an additional word of data for the message) are printed.
- ◆ If the global handle is a queue handle, the queue handle and the chunk handle are printed, with a note that the thing's a queue.
- ◆ If Swat can determine the object's class, the optr, full classname, and current far pointer are printed. In addition, if the chunk has its low bit set, the word "parent" is placed before the output, to denote that the optr likely came from a link and is the parent of the object containing the optr.



See Also: print.

■ fmtval

Usage: fmtval <value-list> <type-token> <indent> [<tail> [<one-line>]]

Examples:

“fmtval [value fetch foo] [symbol find type FooStruct] 0”

Prints the value of the variable foo, which is assumed to be of type FooStruct.

4.2

Synopsis: This is the primary means of producing nicely-formatted output of data in Swat. It is used by both the “print” and “_print” commands and is helpful if you want to print the value of a variable without entering anything into the value history.

Notes:

- ◆ <value-list> is the return value from “value fetch”. You can, of course, construct one of these if you feel so inclined.
- ◆ <type-token> is the token for the type-description used when fetching the value.
- ◆ <indent> is the base indentation for all output. When “fmtval” calls itself recursively, it increases this by 4 for each recursive call.
- ◆ <tail> is an optional parameter that exists solely for use in formatting nested arrays. It is a string to print after the entire value has been formatted. You will almost always omit it or pass the empty string.
- ◆ <one-line> is another optional parameter used almost exclusively for recursive calls. It indicates if the value being formatted is expected to fit on a single line, and so “fmtval” should not force a newline to be output at the end of the value. The value should be 0 or 1.

See Also: print, _print, fntoptr, threadname.

■ focus

Usage: focus [<object>]

Examples:

“focus” print focus hierarchy from the system object down

“focus -i” print focus hierarchy from implied grab down
“focus ^l4e10h:20h”
 print focus hierarchy from ^l4e10h:20h down
“focus [content]”
 print focus hierarchy from content under mouse.

Synopsis: Prints the focus hierarchy below an object.

4.2 **Notes:**

- ◆ If no argument is specified, the system object is used.
- ◆ The special object flags may be used to specify <object>. For a list of these flags, see pobject.
- ◆ Remember that the object you start from may have the focus within its part of the hierarchy, but still not have the focus because something in a different part of the tree has it.
- ◆ The variable “printNamesInObjTrees” can be used to print out the actual app-defined labels for the objects, instead of the class, where available.

 This variable defaults to false.

See Also: target, model, mouse, keyboard, pobject.

■ **focusobj**

Usage: focusobj

Examples:

“focusobj” print model hierarchy from system object down
“pobj [focusobj]”
 Do a pobject on the focus object (equivalent to “pobj -f”).

Synopsis: Returns the object with the focus.

See Also: focus, target, model, targetobj, modelobj.

■ **fonts**

Usage: fonts [<args>]

Examples:



“fonts” summarize general font usage

“fonts -u” list fonts currently in use

Synopsis: Print various font info.

Notes:

◆ The <args> argument may be any of the following:

-a list of fonts available

-d list of font drivers available

-u [<ID>] list of fonts currently in use. Optional font ID to match.

-s summary of above information

4.2

If no argument is specified the default is to show the summary.

◆ When using other commands you probably need to pass them the handle in *FIUE_dataHandle*. When you don't have the font's handle ready, the best way is to use “fonts -u” to find the font at the right point size and then grab the handle from there.

See Also: pfont, pfontinfo, pusage, pchar, pfontinfo.

■ **for** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ **foreach** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ **format** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ **fpstack**

Usage: fpstack

Examples:

 “fpstack” Prints out the hardware and software floating point stacks for the patient.

Synopsis: Prints out the hardware and software floating point stacks for the patient.

See Also: fpstack, pfloat.

■ **fpu-state**

Usage: fpustate [<mode>]



Examples:

“fpustate” Print out the state of the coprocessor.

“fpustate w” Dumps actual words of the numbers.

Synopsis: Prints out the current state of the coprocessor, if any.

See Also: fpstack, pfloat.

4.2 ■ frame

Usage:

```
frame top
frame cur
frame get <ss> <sp> <cs> <ip>
frame next <frame>
frame prev <frame>
frame function [<frame>]
frame funcsym [<frame>]
frame scope [<frame>]
frame info [<frame>]
frame patient [<frame>]
frame register <regName> [<frame>]
frame set [<frame>]
frame setreg <regName> <value> [<frame>]
frame +<number>
frame -<number>
frame <number>
```

Examples:

“var f [frame top]”

Fetches the token for the frame at the top of the current thread's stack and stores it in the variable “f”

“var f [frame next \$f]”

Fetches the token for the next frame up the stack (away from the top) from that whose token is in \$f

“frame register ax \$f”

Returns the value of the AX register in the given frame.

“frame 1”

Sets the current frame for the current thread to be the top-most one.



Synopsis: This command provides access to the stack-decoding functions of swat. Most of the subcommands deal with frame tokens, but a few also handle frame numbers, for the convenience of the user.

Notes:

- ◆ Subcommands may be abbreviated uniquely.
- ◆ Stack decoding works by a heuristic method, rather than relying on the presence of a created stack frame pointed to by BP in each function. Because of this, it can occasionally get confused. 4.2
- ◆ Frame tokens are valid only while the target machine is stopped and are invalidated when it is continued.
- ◆ Each frame records the address on the stack where each register was most-recently pushed (i.e. by the frame closest to it on the way toward the top of the stack). Register pushes are looked for only at the start of a function in what can be considered the function prologue.
- ◆ “frame register” and “frame setreg” allow you to get or set the value held in a register in the given frame. For “setreg”, <value> is a standard address expression, only the offset of which is used to set the register.
- ◆ “frame register” returns all registers but “pc” as a decimal number. “pc” is formatted as two hex numbers (each preceded by “0x”) separated by a colon.
- ◆ “frame info” prints out information on where the register values for “frame register” and “frame setreg” are coming from/going to for the given or currently-selected frame. Because of the speed that can be gained by only pushing registers when you absolutely have to, there are many functions in GEOS that do not push the registers they save at their start, so Swat does not notice that they are actually saved. It is good to make sure a register value is coming from a reliable source before deciding your program has a bug simply because the value returned by “frame register” is invalid.
- ◆ For any subcommand where the <frame> token is optional, the currently-selected frame will be used if you give no token.
- ◆ “frame cur” returns the token for the currently-selected stack frame.
- ◆ “frame set” is what sets the current frame, when set by a Tcl procedure.

4.2

- ◆ “frame +<number>” selects the frame <number> frames up the stack (away from the top) from the current frame. “frame -<number>” goes the other way.
- ◆ “frame <number>” selects the frame with the given number, where the top-most frame is considered frame number 1 and numbers count up from there.
- ◆ “frame funcsym” returns the symbol token for the function active in the given (or current) frame. If no known function is active, you get “nil”.
- ◆ “frame scope” returns the full name of the scope that is active in the given (or current) frame. This will be different from the function if, for example, one is in the middle of an “if” that contains variables that are local to it only.
- ◆ “frame function” returns the name of the function active in the given (or current) frame. If no known function is active, you get the CS:IP for the frame, formatted as two hex numbers separated by a colon.
- ◆ “frame patient” returns the token for the patient that owns the function in which the frame is executing.

See Also: addr-parse, switch.

■ framewin

Usage: framewin [del]

Examples:

“framewin”

Creates a single-line window to display info about the current stack frame.

“framewin del”

Deletes the window created by a previous “framewin”.

Synopsis: Creates a window in which the current stack frame is always displayed.

Notes:

- ◆ Only one frame window can be active at a time.

See Also: display, regwin, ewatch, srcwin



■ freeze

Usage: freeze [<patient>]
freeze :<n>
freeze <patient>:<n>
freeze <id>

Examples:

“freeze” Freezes the current thread. 4.2
“freeze term”
 Freezes the application thread for “term”
“freeze :1” Freezes thread #1 of the current patient
“freeze 16c0h”
 Freezes the thread whose handle is 16c0h.

Synopsis: Freezing a thread prevents a thread from running unless it's the only thread that's runnable in the entire system.

Notes:

- ◆ A frozen thread is not dead in the water, as it will still run if nothing else is runnable.
- ◆ Freezing a thread is most useful when debugging multi-threaded applications where a bug appears to be caused by a timing problem or race condition between the two threads. Freezing one of the threads ensures a consistent timing relationship between the two threads and allows the bug to be reproduced much more easily.
- ◆ The freezing of a thread is accomplished by setting its base and current priorities to as high a number as possible (255) thereby making the thread the least-favored thread in the system. The previous priority can be restored using the “thaw” command.

See Also: thaw.

■ fullscreen

Usage: fullscreen

Examples:

“fullscreen”

Synopsis: Prints the full screen hierarchy from the system object down.

■ func

Usage: func [<func name>]

Examples:

“func” return the current function.

“func ObjMessage”
set the frame to the first frame for ObjMessage.

Synopsis: Get the current function or set the frame to the given function.

Notes:

- ◆ The <func name> argument is the name of a function in the stack frame of the current patient. The frame is set to the first occurrence of the function from the top of the stack.
- ◆ If no <func name> argument is given then ‘func’ returns the current function.

See Also: backtrace, up, down, finish.

■ fvardata

Usage: fvardata <token> [<address>]

Examples:

“fvardata ATTR_VIS_TEXT_STYLE_ARRAY *ds:si”

Synopsis: Locates and returns the value list for the data stored under the given token in the vardata of the given object.

Notes:

- ◆ If the data are found, returns a list {<token> <data>}, where <data> is a standard value list for the type of data associated with the specified token.
- ◆ Returns an empty list if the object has no vardata entry of the given type.
- ◆ If no <address> is given, the default is *ds:si.



■ fwalk

Usage: fwalk [<patient>]

Examples:

“fwalk” list all open files.

“fwalk geos” list all open files owned by the GEOS patient.

4.2

Synopsis: Print the list of files open anywhere in the system.

Notes:

- ◆ The patient argument may be used to restrict the list to a particular patient. The patient may be specified either as the patient name or as the patient's handle.
- ◆ fwalk differs from **sysfiles** and **geosfiles** in that it deals primarily with GEOS data structures.
- ◆ The 'Other' column shows if there is a VM handle bound to the file.
- ◆ The letters in the 'Flags' column mean the following:

RW deny RW

R deny R

W deny W

N deny none

rw access RW

r access R

w access RW

O override, used to override normal exclusion normally used by **FileEnum()** to check out file headers.

E exclusive, used to prevent override. This is used by **swap.geo**

See Also: fhandle, geosfiles, sysfiles.

■ gc

Usage: gc [(off | register | <extensive-heap-checking-flag>)]

Synopsis: Implements a simple garbage collector to scavenge unreferenced symbols and types. If given an argument other than “off” or “register,” it turns on extensive heap checking, which slows things down enormously but ensures the heap is in good shape. The “gc register” command can be used to register a type created by “type make” as something that is being used for an extended period at the Tcl level, preventing the thing from being garbage-collected.

■ gentree

4.2

Usage: gentree [<address>] [<instance field>]

Examples:

```
“gentree”      print the generic tree starting at *DS:SI
“gentree -i”   print the generic tree under the mouse
“gentree [systemobj]”
                print the generic tree starting at the system’s root
“gentree @23 GI_states”
                print the generic tree with generic states
“gentree *uiSystemObj”
                start the generic tree at the root of the system
```

Synopsis: Print a generic tree.

Notes:

- ◆ The <address> argument is the address to an object in the generic tree. This defaults to *DS:SI. The ‘-i’ flag for an implied grab may be used.
- ◆ The special object flags may be used to specify <object>. For a list of these flags, see pobject.
- ◆ The <instance field> argument is the offset to any instance data within the GenInstance which should be printed out.
- ◆ The variable “printNamesInObjTrees” can be used to print out the actual app-defined labels for the objects, instead of the class, where available. This variable defaults to false.

See Also: gup, vistree, impliedgrab, systemobj, pobject.



■ geosfiles

Usage: geosfiles

Examples: “geosfiles”

Synopsis: Print out all the files for which I/O is currently pending in GEOS.

Notes:

- ◆ This looks at the same dos structure as sysfiles but this prints only those files also listed in GEOS’ job file table.

4.2

See Also: sysfiles, sftwalk, fwalk.

■ geos-release

Synopsis: This variable contains the major number of the version of GEOS running on the target PC.

■ geowatch

Usage: geowatch [<object>]

Examples:

“geowatch *MyObj”

Display geometry calls that have reached the object MyObj

“geowatch” Display geometry calls that have reached *ds:si (asm) or oself (goc)

Synopsis: This displays geometry calls that have reached a particular object. Only one object at a time can be watched in this way.

Notes:

- ◆ Two conditional breakpoints are used by this function (see cbrk). The tokens for these breakpoints are returned.
- ◆ The special object flags may be used to specify *object*. For a list of these flags, see pobject.

See Also: objwatch, mwatch, cbrk, pobject.

■ get-address

Used by the various memory-access commands. Takes one argument, ADDR, being the address argument for the command. Typically, the command is declared as

```
[defcmd cmd {{addr nil}} ... ]
```

4.2

allowing the address to be unspecified. This function will return the given address if it was, else it will return the last-accessed address (stored in the global *lastAddr* variable as a 3-tuple from *addr-parse*) in the form of an address expression. If no address is recorded (*lastAddr* is nil), the default-addr argument is used. If it is not specified then CS:IP will be used.

■ getcc

Usage: getcc <flag>

Examples:

“getcc c” Get the carry flag.

Synopsis: Get a flag from the target machine.

Notes:

- ◆ The first argument is the first letter of the flag to get. The following is a list of the flags:

t	trap
i	interrupt enable
d	direction
o	overflow
s	sign
z	zero
a	auxiliary carry
p	parity
c	carry

- ◆ This command is handy to run with a breakpoint to stop if a flag is set.



See Also: setcc, clrcc, compcc.

■ getenv

Usage: getenv <NAME>

Examples:

“getenv PTTY”

Fetches the value of the host machine’s PTTY environment variable.

4.2

Synopsis: Returns the value for a variable defined in Swat’s environment.

Notes: If the variable isn’t defined, this returns the empty string.

See Also: var, string.

■ get-key-binding

Usage: get-key-binding <char>

Examples:

“get-key-binding c”

Gets key binding for the character c.

“get-key-binding \045”

Gets key binding for the % key.

Synopsis: Gets key binding for given key.

See Also: alias, bind-key, unbind-key.

■ getvalue

Usage: getvalue <expr>

Examples:

“getvalue MSG_META_DETACH”

Returns the integer value of the symbol MSG_META_DETACH.

Synopsis: This is a front-end to the “addr-parse” command that allows you to easily obtain the integer value of any expression. It’s most useful for converting something the user might have given you to a decimal integer for further processing.

Notes: If the expression you give does not evaluate to an address (whose offset will be returned) or an integer, the results of this function are undefined.

See Also: addr-parse, addr-preprocess.

■ **global** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ **go**

4.2

Usage: go [<address expressions>]

Examples:

“go” “go drawLeftLine”

Synopsis: Go until an address is reached.

Notes:

- ◆ The <address expressions> argument is as many address expressions as desired for breakpoints. Execution is continued until a breakpoint is reached. These breakpoints are then removed when the machine stops and are only active for the current patient.

See Also: break, continue, det, quit.

■ **grobjtree**

Usage: grobjtree [<address>] [<instance field>]

Examples:

“grobjtree” Print the grobj tree starting at *ds:si

Synopsis: Print out a GrObj tree.

Notes:

- ◆ The address argument is the address of a GrObj Body This defaults to *ds:si.
- ◆ To get the address of the grobj body, use the “pbody” or “target” commands.

See Also: pbody.



■ gup**Usage:** gup [<address>] [<instance field>]**Examples:**

“gup” print the generic object at *DS:SI and its ancestors

“gup @23 GI_states”
 print the states of object @23 and its ancestors

“gup -i” print the generic object under the mouse and the object’s
 ancestors

4.2

Synopsis: Print a list of the object and all of its generic ancestors.**Notes:**

- ◆ The address argument is the address to an object in the generic tree. This defaults to *DS:SI. The ‘-i’ flag for an implied grab may be used.
- ◆ The special object flags may be used to specify <object>. For a list of these flags, see pobject.
- ◆ The instance field argument is the offset to any instance data within the GenInstance which should be printed out.

See Also: gentree, vup, vistree, impliedgrab.**■ handle** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.**■ handles****Usage:** handles [<flags>] [<patient>]

Examples: “handles”
 “handles -f”
 “handles ui”

Synopsis: Print all handles in-use.**Notes:**

- ◆ The flags argument is a collection of flags, beginning with ‘-’, from the following set:
 - s** print summary only.

- e** events only.
- p** don't print prevPtr and nextPtr.
- f** fast print-out - this doesn't try to figure out the block type.
- r** reverse, i.e. starts at the end of the handle table.
- u** print only those handles that are in-use.

4.2

- ◆ The *patient* argument is a patient whose blocks are to be selectively printed (either a name or a core-block's handle ID). The default is to print all the blocks on the heap.

- ◆ The following columns can appear in a listing:

HANDLE	The handle of the block
ADDR	The segment address of the block
SIZE	Size of the block in bytes
PREV	The previous block handle (appears with the p flag)
NEXT	The next block handle (appears with the p flag)
FLAGS	The following letters appears in the FLAGS column: s sharable, S swapable, D discardable, L contains local memory heap, d discarded (by LMem module: discarded blocks don't appear here), a attached (notice given to swat whenever state changes)
LOCK	Number of times the block is locked or n/a if FIXED.
OWNER	The process which owns the block
IDLE	The time since the block has been accessed in minutes:seconds
OINFO	The <i>otherInfo</i> field of the handle (block type dependent)
TYPE	Type of the block, for example: R#1 (dgroup) resource number one

- ◆ This only prints those handles in memory while 'handles' prints all handles used.
- ◆ The handles may be printed with lhwalk and phandle.



See Also: lhwalk, phandle, hgwalk.

■ handsum

Usage: handsum

Examples:

“handsum” Summarize the use to which the handle table is being put.

Synopsis: This command analyzes the handle table and prints out a list of the number of handles being used by each geode, and for what purpose. 4.2

Notes:

- ◆ The columns of the output are labeled somewhat obscurely, owing to horizontal-space constraints. The headings, and their meanings are:

Res	Resource handles (i.e. handles for data stored in the geode’s executable)
Mem	Non-resource memory handles
File	Open files
Thds	Threads
Evs	Recorded events
Qs	Event queues
Sems	Semaphores
EDat	Data for recorded events
Tim	Timers
SB	Saved blocks (handles tracking memory/resource handles whose contents will go to an application’s state file)
VMF	VM files

- ◆ The “handles” command is good at printing out all the handles for a particular geode, but it’s generally too verbose to use for the entire handle table. That’s why this command exists.

- ◆ It’s a good idea to issue the command “dcache length 4096” before executing this command, as it ensures the entire handle table will end up

in Swat's data cache, for quick access if you want to use the "handles" command immediately afterward.

■ hbrk

Usage: hbrk <address> (byte | word) (match | mismatch) <value>

Examples:

4.2

"hbrk scrollTab+10 byte match 0"
print message handlers until a zero is written at scrollTab+10.

"hbrk OLScrollButton+3 word mismatch 0x654f"
Break when the word at OLScrollButton+3 is destroyed.

Synopsis: Break when a memory location changes.

Notes:

- ◆ The <address> argument is the address to watch for a change.
- ◆ The (byte | word) argument indicates whether to watch a byte or a word for a change.
- ◆ The (match | mismatch) argument indicates whether to break if the value at the address matches or mismatches the value hbrk is called with.
- ◆ hbrk emulates a hardware breakpoint by checking at every message call to see if a location in memory has been written to. If so, swat breaks and tells between which two messages the write occurred. The information and the return stack will hopefully guide you to the offending line of code.
- ◆ The command creates two breakpoints. Remove these to get rid of the hardware breakpoint.

See Also: brk, mwatch, showcalls.

■ heapSPACE

Usage: heapSPACE <geode>
heapSPACE total
heapSPACE syslib

Examples:

"heapSPACE geomanager"
print out "heapSPACE" value for geomanager



“heap space total”

print out maxTotalHeapSpace

“heap space syslib”

print out space being used by system libraries.

Synopsis:

Prints out how much space the program requires on the heap. This value may then be used in a “heap space” line of the program’s .gp field. This command only determines present usage—to determine the most heap space your geode will ever use requires that you make it allocate as much space as it ever will. This means pulling down all menus, opening all dialog boxes, and generally building out all UI gadgetry. The value this command prints is roughly the non-discardable heap usage by the app and any transient libraries that it depends on, plus an additional amount for thread activity.

4.2

■ help

Usage: help [<command>]

Synopsis: This is the user-level access to the on-line help facilities for Swat. If given a topic (e.g. “brk”) as its argument, it will print all help strings defined for the given topic (there could be more than one if the same name is used for both a variable and a procedure, for instance). If invoked without arguments, it will enter a browsing mode, allowing the user to work his/her way up and down the documentation tree.

■ help-fetch

Usage: help-fetch <topic-path>

Examples: “help-fetch top.patient”

Synopsis: Fetches the help string for a given topic path in the help tree.

Notes: If there is more than one node with the given path in the help tree, only the string for the first node will be returned.

■ help-fetch-level

Usage: help-fetch-level

Examples:

“help-fetch-level top.prog.obscure”

Returns the topics within the “top.prog.obscure” level of the help tree.

Synopsis: Returns a list of the topics available at a given level in the help tree.

Notes: The result is a list of node names without leading path components.

See Also: help-fetch.

4.2

■ help-help

Usage: help-help

Synopsis: Provides help about using the help command (q.v.)

See Also: help.

■ help-is-leaf

Usage: help-is-leaf <topic-path>

Examples:

“help-is-leaf top.prog”

See if top.prog is a leaf node in the help tree (i.e. if it has no children).

Synopsis: Determines whether a given path refers to a help topic or a help category.

Notes: Returns one if the given path refers to a leaf node, zero if it is not.

See Also: help-fetch, help-fetch-level.

■ help-minAspect

Usage: var help-minAspect [<ratio-times-ten>]

Synopsis: If non-zero, contains the minimum aspect ratio to be maintained when displaying tables in the help browser. The ratio is expressed as the fraction

$$\text{entries_per_column} * 10 / \text{number_of_columns}$$

E.g. a minimum ratio of 1.5 would be 15. (We multiply by ten because Swat doesn't support floating point numbers.)



■ help-scan

Usage: help-scan <pattern>

Examples:

“help-scan break”

Looks for all nodes at any level of the help tree whose documentation includes the pattern “break”.

Synopsis: Scans all nodes in the help tree for those whose documentation matches a given pattern.

4.2

Notes:

◆ The result is a list of topic-paths.

See Also: help-fetch.

■ help-verbose

Usage: var help-verbose [0 | 1]

Synopsis: If non-zero, performs verbose prompting.

■ hex

Usage: hex <number>

Examples:

“hex 034” print hex equivalent of octal 34.

“hex 12” print hex equivalent of decimal 12.

Synopsis: Print hexadecimal equivalent of a number.

■ hgwalk

Usage: hgwalk

Examples:

“hgwalk” print statistics on all geodes

Synopsis: Print out all geodes and their memory usage.

■ history

Usage: history [<args>]

Examples:

“history 10”

Prints the last 10 commands entered via the “history subst” command.

4.2

“history subst \$line”

Performs history substitution on the string in \$line, enters the result in the history queue and returns the result.

“var n [history cur]”

Stores the number of the next string to be entered via “history subst” in the variable n.

“history set 50”

Limit the number of entries in the queue to 50.

“history fetch 36”

Returns the string entered as command number 36 in the history queue.

Synopsis:

This command manipulates the history list. Options are:

<number> Prints the most-recent <number> commands

set <queue-size>

Sets the number of commands saved

subst <str> Performs history substitution on <str> and enters it into the history queue.

cur Returns the current history number. If no argument is given, all saved commands are printed.

fetch <n> Returns the string entered as command number <n> in the history queue.

See Also:

top-level-read

■ hwalk

Usage: hwalk [<flags>] [<patient>]

Examples:



“hwalk” display the heap
 “hwalk -e” display the heap and perform error checking
 “hwalk -r ui” display the heap owned by the ui in reverse order

Synopsis: Print the status of all blocks on the global heap.

Notes:

◆ The <flags> argument is a collection of flags, beginning with ‘-’, from the following set:

4.2

r print heap in reverse order (decreasing order of addresses)
p print prevPtr and nextPtr as well.
e do error-checking on the heap.
l just print out locked blocks
f fast print-out—this doesn’t try to figure out the block type
F print out only fixed (or pseudo-fixed) resources.
c Print out only code resources (discardable or fixed non-lmem non-dgroup resources).
s <num> start at block <num>

◆ The patient argument is a patient whose blocks are to be selectively printed (either a name or a core-block’s handle ID). The default is to print all the blocks on the heap.

◆ The following columns can appear in a listing:

HANDLE	The handle of the block
ADDR	The segment address of the block
SIZE	Size of the block in bytes
PREV	The previous block handle (appears with the p flag)
NEXT	The next block handle (appears with the p flag)
FLAGS	The following letters appears in the FLAGS column:
	s sharable
	S swapable

Tools 

Swat Reference

148

	D discardable
	L contains local memory heap
	d discarded (by LMem module: discarded blocks don't appear here)
	a attached (notice given to swat whenever state changes)
	LOCK Number of times the block is locked or n/a if FIXED.
	OWNER The process which owns the block
4.2	IDLE The time since the block has been accessed in minutes:seconds
	OINFO The otherInfo field of the handle (block type dependent)
	TYPE Type of the block, for example: R#1 (dgroup) Resource number one, named "dgroup" Geode Internal control block for a geode WINDOW, GSTATE, Internal structures of the given type GSTRING, FONT_BLK, FONT OBJ(write:0) Object block run by thread write:0 VM(3ef0h)... VM block from VM file 3ef0h
	◆ This only prints those handles in memory while 'handles' prints all handles used.
	◆ The handles may be printed with lhwalk and phandle.
See Also:	lhwalk, phandle, handles, hgwalk.

■ iacp

Usage:

iacp -ac
prints all connections

iacp -l
prints all lists without connections

iacp -d
prints all open documents

iacp <obj>
prints all connections to which <obj> is party



■ ibrk

Set a breakpoint interactively. At each instruction, you have several options:

q	Quit back to the command level.	
n	Go to next instruction (this also happens if you just hit return).	
p	Go to previous instruction.	
P	Look for a different previous instruction.	4.2
^D	Go down a “page” of instructions. The size of the page is controlled by the global variable <code>ibrkPageLen</code> . It defaults to 10.	
^U	Go up a “page” of instructions.	
b	Set an unconditional breakpoint at the current instruction and go back to command level.	
a	Like 'b', but the breakpoint is set for all patients.	
t	Like 'b', except the breakpoint is temporary and will be removed the next time the machine stops.	
B	Like 'b', but can be followed by a command to execute when the breakpoint is hit.	
A	Like 'B', but for all patients.	
T	Like 'B', but breakpoint is temporary.	

■ ibrkPageLen

Usage: `var ibrkPageLen [<number-of-lines>]`

Synopsis: Number of instructions to skip when using the `^D` and `^U` commands of `ibrk`.

■ if This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ ignerr

Usage: `ignerr`

Examples:

“ignerr” ignore error and continue

“ignerr MyFunc::done” ignore error and continue at MyFunc::done.

Synopsis: Ignore a fatal error and continue.

Notes:

- ◆ The address argument is the address of where to continue execution. If not specified then CS:IP is taken from the frame.
- ◆ The stack is patched so that execution can continue in the frame above the fatal error handling routine.

4.2 **See Also:** why, backtrace.

■ imem

Usage: imem [<address>] [<mode>]

Examples:

“imem” enter imem mode at DS:SI

“imem ds:di”
 enter imem mode at DS:SI

Synopsis: Examine memory and modify memory interactively.

Notes:

- ◆ The address argument is the address to examine. If not specified, the address after the last examined memory location is used. If no address has been examined then DS:SI is used for the address.
- ◆ The mode argument determines how the memory is displayed and modified. Each of the four modes display the memory in various appropriate formats. The modes are:

Table 4-1 *Memory Modes*

Mode	Size	1st column	2nd column	3rd column
b	byte	hex byte	signed dec	ASCII character
w	word	hex word	unsigned dec	signed decimal
d	dword	segment:offset	signed dec	symbol
i	???	hex bytes	assembler instr.	

- ◆ The default mode is swat’s best guess of what type of object is at the address.



- ◆ **imem** lets you conveniently examine memory at different locations and assign it different values. **imem** displays the memory at the current address according to the mode. From there you can move to another memory address or you can assign the memory a value.

- ◆ You may choose from the following single-character commands:

b, w, d, i Sets the mode to the given one and redisplay the data.

n, j, <Return>

Advances to the next data item. The memory address advances by the size of the mode.

4.2

p, k

Returns to the preceding data item. The memory address decreases by the size of the mode. When displaying instructions, a heuristic is applied to locate the preceding instruction. If it chooses the wrong one, use the 'P' command to make it search again.

<space>

Clears the data display and allows you to enter a new value appropriate to the current display mode. The "assign" command is used to perform the assignment, so the same rules apply to it, with the exception of ' and " -quoted strings. A string with 's around it ('hi mom') has its characters poked into memory starting at the current address. A string with "s around it ("swat.exe") likewise has its characters poked into memory, with the addition of a null byte at the end. This command is not valid in instruction mode.

q

quit **imem** and return to command level. The last address accessed is recorded for use by the other memory-access commands.

^D

Display a "page" of successive memory elements in the current mode.

^U

Display a "page" of preceding memory elements in the current mode.

h, ?

This help list.

For **^D** and **^U**, the size of a "page" is kept in the global variable *imemPageLen*, which defaults to 10.

See Also:

bytes, words, dwords, assign.



■ imemPageLen

Usage: var imemPageLen [<numlines>]

Synopsis: Contains the number of elements to display when imem is given the ^D or ^U command.

■ impliedgrab

4.2

Usage: impliedgrab

Examples:

“gentree [impliedgrab]”
print the generic tree under the mouse

Synopsis: Print the address of the current implied grab, which is the screen object grabbing the mouse.

Notes:

- ◆ This command is normally used with gentree to get the generic tree of an application by placing the mouse on application's window and issuing the command.

See Also: systemobj, gentree.

■ impliedwin

Usage: impliedwin

Notes:

“wintree [impliedwin]”
print the window tree of the window under the mouse

Synopsis: Print the address of the current implied window (the window under the mouse).

Notes:

- ◆ Note that a window handle is returned.
- ◆ This command is normally used with wintree. One may also use the print command if they properly cast the handle.



■ **index** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ **info** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ **int**

Usage: int [<int level> <state>]

Examples:

“int” report the interrupt statuses

4.2

“int 1:1 on” allow keyboard interrupt while in swat

Synopsis: Set or print the state of the two interrupt controllers for when then machine is stopped in Swat.

Notes:

- ◆ If no arguments are given, the current state is printed.
- ◆ The <int level> argument is specified by their names or their numbers with the form <controller>:<number>. <controller> is either 1 or 2, and <number> ranges from 0 to 7. The interrupts and their numbers are:

Timer 1:0 System timer. Probably dangerous to enable.

Keybd 1:1 Keyboard input.

Slave 1:2 This is how devices on controller 2 interrupt. Disabling this disables them all.

Com2 1:3 This is the port usually used by Swat, so it can't be disabled.

Com1 1:4 The other serial port -- usually the mouse.

LPT2 1:5 The second parallel port

Floppy 1:6 Floppy-disk drive

LPT1 1:7 First parallel port

Clock 2:0 Real-time clock

Net 2:1 Network interfaces (?)

FPA 2:5 Coprocessor

HardDisk 2:6

Hard-disk drive

- ◆ The <state> argument is either on or off.

■ intFormat

Usage: var intFormat [<format-string>]

Examples:

“var intFormat %d”

Sets the default format for printing unsigned integers to decimal.

Synopsis: *\$intFormat* contains the string passed to the “format” command to print an integer.

Notes: The default value is {*%xh*}, which prints the integer in hexadecimal, followed by an “h”.

See Also: print, byteAsChar.

■ intr

Catch, ignore, or deliver an interrupt on the target PC. First argument is the interrupt number. Optional second argument is “catch” to catch delivery of the interrupt, “ignore” to ignore the delivery, or “send” to send the interrupt (the machine will keep going once the interrupt has been handled). If no second argument is given, the interrupt is delivered.

■ io

Usage: io [w] <port> [<value>]

Examples:

“io 21h” Reads byte-sized I/O port 21h.

“io 20h 10” Writes decimal 10 to byte-sized I/O port 20h.

Synopsis: Provides access to any I/O port on the PC.

Notes:



- ◆ If you give the optional first argument “w” then Swat will perform a 16-bit I/O read or write, rather than the default 8-bit access. Be aware that most devices don’t handle this too well.
- ◆ <port> must be a number (in any radix); it cannot be a register or other complex expression.
- ◆ If you don’t give a <value>, you will be returned the contents of the I/O port (it will not be printed to the screen).

■ irq

4.2

Usage: irq
 irq (no | yes)
 irq (set | clear)

Examples:

“irq” Returns non-zero if an interrupt is pending.
“irq no” Disable recognition and acting on a break request from the keyboard.
“irq set” Pretend the user typed Ctrl-C.

Synopsis: Controls Swat’s behavior with respect to interrupt requests from the keyboard.

Notes:

- ◆ Swat maintains an interrupt-pending flag that is set when you type Ctrl+C (it can also be set or cleared by this command). It delays acting on the interrupt until the start of the next or the completion of the current Tcl command, whichever comes first.
- ◆ When given no arguments, it returns the current state of the interrupt-pending flag. This will only ever be non-zero if Swat is ignoring the flag (since the command wouldn’t actually return if the flag were set and being paid attention to, as the interpreter would act on the flag to vault straight back to the command prompt).
- ◆ If given “no” or “yes” as an argument, it causes Swat to ignore or pay attention to the interrupt-pending flag, respectively.
- ◆ You can set or clear the flag by giving “set” or “clear” as an argument.

■ is-obj-in-class

Usage: is-obj-in-class <obj-addr> <class-name>

Examples:

“is-obj-in-class ^14e10h:1eh GenPrimaryClass”
see if the object at ^14e10h:1eh is in **GenPrimaryClass**.

4.2

Synopsis: Returns whether a given object in the specified class.

Notes:

- ◆ Returns one if the object is in the specified class, zero otherwise. It will return one if the object's class is a subclass of the passed class.
- ◆ The special object flags may be used to specify <object>. For a list of these flags, see pobject.

See Also: psup.

■ istep

Usage: istep [<default command>]

Examples:

“is” enter instruction step mode
“istep n” enter instruction step mode, <ret> does a next command

Synopsis: Step through the execution of the current patient. This is THE command for stepping through assembly code.

Notes:

- ◆ The default command argument determines what pressing the <Return> key does. By default, <Return> executes a step command. Any other command listed below may be substituted by passing the letter of the command.
- ◆ Istep steps through the patient instruction by instruction, printing where the ip is, what instruction will be executed, and what the instruction arguments contain or reference. Istep waits for the user to type a command which it performs and then prints out again where istep is executing.
- ◆ This is a list of **istep** commands:



q, <Esc>, ‘‘	Stops istep and returns to command level.	
b	Toggles a breakpoint at the current location.	
c	Stops istep and continues execution.	
n	Continues to the next instruction, skipping procedure calls, repeated string instructions, and software interrupts. Using this procedure, istep only stops when the machine returns to the right context (i.e. the stack pointer and current thread are the same as they are when the “n” command was given). Routines which change the stack pointer should use “N” instead.	4.2
o	Like “n” but steps over macros as well.	
l	Goes to the next library routine.	
N	Like ‘n’, but stops whenever the breakpoint is hit, whether you’re in the same frame or not.	
O	Like ‘N’ but steps over macros as well.	
m, M	Goes to the next method called. Doesn’t work when the message is not handled anywhere.	
F	Finishes the current message.	
f	Finishes out the current stack frame.	
s, <Return>	Steps one instruction.	
A	Aborts the current stack frame.	
S	Skips the current instruction	
B	Backs up an instruction (opposite of “S”).	
J	Jump on a conditional jump, even when “Will not jump” appears. This does not change the condition codes.	
g	Executes the ‘go’ command with the rest of the line as arguments.	
e	Executes a Tcl command and returns to the prompt.	
r	Lists the registers (uses the regs command)	
R	References either the function to be called or the function currently executing.	

Swat Reference

158

h, ? Displays a help message.

- ◆ If the current patient isn't the actual current thread, **istep** waits for the patient to wake up before single-stepping it.

See Also: sstep, listi, ewatch.

4.2



■ keyboard

Usage: keyboard [<object>]

Examples:

“keyboard” print keyboard hierarchy from system object down

“keyboard -i” print keyboard hierarchy from implied grab down

“keyboard ^l4e10h:20h”
print keyboard hierarchy from ^l4e10h:20h down.

Synopsis: Prints the keyboard hierarchy below an object.

Notes:

- ◆ If no object is specified, the system object is used.
- ◆ The special object flags may be used to specify <object>. For a list of these flags, see pobject.
- ◆ The variable “printNamesInObjTrees” can be used to print out the actual app-defined labels for the objects, instead of the class, where available. This variable defaults to false.

See Also: target, focus, mouse, model, pobject.

■ keyboardobj

Usage: keyboardobj

Examples:

“keyboardobj”
return object with keyboard grab

“pobj [keyboardobj]”
do a **pobject** on the object with the keyboard grab (equivalent to “pobj -kg”).

Synopsis: Returns the object with the keyboard grab.

See Also: target, focus, mouse, keyboard, mouseobj.

■ lastCommand

Usage: `$lastCommand`

Examples:

`"var repeatCommand $lastCommand"`

Set the current command as the command to execute next time.

Synopsis: `$lastCommand` stores the text of the command currently being executed.

Notes: This variable is set by `top-level-read`. Setting it yourself will have no effect, unless you call `set-address` or some similar routine that looks at it.

See Also: `repeatCommand`, `top-level-read`.

■ **length** This is a Tcl primitive. See "Tool Command Language," Chapter 5.

■ lhwalk

Usage: `lhwalk [<address>]`

Examples:

`"lhwalk 1581h"`

list the lm heap at 1581h:0

Synopsis: Prints out information about a local memory heap.

Notes:

- ◆ The address argument is the address of the block to print. The default is the block pointed to by DS.

See Also: `hwalk`, `objwalk`

■ link

Usage: `link <library> [<patient>]`

Examples:

`"link motif"` Makes the library "motif" a library of the current patient as far as Swat is concerned.

Synopsis: Allows you to link a patient to act as an imported library of another patient, even though the other patient doesn't actually import the patient. This is useful only for symbol searches.



Notes:

- ◆ `sym-default` is a much better way to have Swat locate symbols for libraries that are loaded by **`GeodeUseLibrary()`**.
- ◆ Cycles are not allowed. I.e. don't link your application as a library of the UI, as it won't work—or if it does, it will make Swat die.
- ◆ The link persists across detach/attach sequences so long as the `<patient>` isn't recompiled and downloaded.
- ◆ If you don't give `<patient>`, then the current patient will be the one made to import `<library>`
- ◆ Both `<library>` and `<patient>` are patient *names*, not tokens.

See Also: `help-fetch`.

■ **list** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ **listi**

Usage: `listi [<address>] [<length>]`

Examples:

```

“1”           disassemble at the current point of execution
“listi geos::Dispatch”
                disassemble at the kernel's dispatch routine
“listi DocClip:IsOut”
                disassemble at the local label
“listi cs:ip 20”
                disassemble 20 instructions from the current point of execution

```

Synopsis: Disassemble at a memory address.

Notes:

- ◆ The `<address>` argument is the address to examine. If not specified, the address after the last examined memory location is used. If no address has been examined then CS:IP is used for the address.
- ◆ The `<length>` argument is the number of instructions to list. It defaults to 16.
- ◆ Pressing `<Return>` after this command continues the list.



See Also: istep, sstep, skip, where.

■ load

Usage: load <file>

Synopsis: Load a file of Tcl commands.

- ◆ If the file cannot be found as given, it is looked for in all the directories mentioned in the “load-path” variable. This variable is initialized from the SWATPATH environment variable, which is in the form <dir1>:<dir2>:...:<dirN>.
 - ◆ The Swat library directory is appended to this path so you need not include it yourself. The file need not end in “.tcl”.
 - ◆ When searching, *file*, *file.tcl*, and *file.tlc* are searched for. If **load** finds a *file.tlc* file, that file will be used only if it is more recent than any corresponding *file.tcl* or *file* file.
-

■ loadapp

Load an application from swat. Single argument is the file name of the application to launch (application must reside in the appl subdirectory of the GEOS tree).

The application is opened in normal application mode. Note that the application will not be loaded until you continue the machine, as the loading is accomplished by sending a message to the UI.

■ loadgeode

Load a geode from swat. Mandatory first argument is the name of the file to load (with path from top-level GEOS directory, using / instead of \ as the path separator).

Second and third arguments are the data words to pass to the geode. The second argument is passed to the geode in cx, while the third argument is passed in dx.

Both the second and third arguments are optional and default to 0. They likely are unnecessary.



■ locals

Usage: locals [<func>]

Examples:

“locals” Print the values of all local variables and arguments for the current frame.

“locals WinOpen”
 Print the names of all local variables for the given function. No values are printed.

Synopsis: Allows you to quickly find the values or names of all the local variables of a function or stack frame.

See Also: print, frame info

■ localwin

Usage: localwin [<numlines>]

Examples:

“localwin” Display local variables in a 10-line window

“localwin 15”
 Display local variables in a 15-line window

“localwin off”
 Turn off the local variable display

Synopsis: Turn on or off the continuous display of local variables.

Notes:

- ◆ Passing an optional numerical argument turns on display of that size. The default size is 10 lines.
 - ◆ Only one local variable display may be active at a time.
-

■ loop

Simple integer loop procedure. Usage is:

loop <loop-variable> <start>,<end> [step <step>] <body>

<start>, <end>, and <step> are integers. <body> is a string for Tcl to evaluate. If no <step> is given, 1 or -1 (depending as <start> is less than or



greater than <end>, respectively) is used. <loop-variable> is any legal Tcl variable name.

■ map

Usage: map <var-list> <data-list>+ <body>

Examples:

“map {i j} {a b} {c d} {list \$i \$j}”

Executes the command “list \$i \$j” with i and j assigned to successive elements of the lists {a b} and {c d}, respectively, merging the results into the list {{a c} {b d}}

Synopsis: This applies a command string to the successive elements of one or more lists, binding each element in turn to a variable and evaluating the command string. The results of all the evaluations are merged into a result list.

Notes:

- ◆ The number of variables given in <var-list> must match the number of <data-list> arguments you give.
- ◆ All the <data-list> arguments must have the same number of elements.
- ◆ You do not specify the result of the <body> with the “return” command. Rather, the result of <body> is the result of the last command executed within <body>.

See Also: foreach, mapconcat.

■ mapconcat

Usage: mapconcat <var-list> <data-list>+ <body>

Examples:

“mapconcat {i j} {a b} {c d} {list \$i \$j}”

Executes the command “list \$i \$j” with i and j assigned to successive elements of the lists {a b} and {c d}, respectively, merging the results into a string.



■ map-method

Usage: map-method <number> <object>
 map-method <number> <class-name> [<object>]

Examples:

“map-method ax ^lbx:si”
 Prints the name of the message in ax, from the object at
 ^lbx:si’s perspective.

“map-method 293 GenClass”
 Prints the name of message number 293 from GenClass’s
 perspective.

Synopsis: Maps a message number to a human-readable message name, returning that
 name. This command is useful both for the user and for a Tcl procedure.

Notes:

- ◆ When called from a Tcl procedure, the <class-name> argument should be the fullname of the class symbol (usually obtained with the obj-class function), and <object> should be the address of the object for which the mapping is to take place. If no <object> argument is provided, map-method will be unable to resolve messages defined by one of the object’s superclasses that lies beyond a variant superclass.
- ◆ If no name can be found, the message number, in decimal, is returned.
- ◆ The result is simply returned, not echoed. You will need to echo the result yourself if you call this function from anywhere but the command line.

See Also: obj-class.

■ mcount

Usage: mcount [<args>]

Examples:

“mcount” start the method count or print the count

“mcount reset”
restart the method count

“mcount stop”
stop the method count

“mcount MyAppRecalcSize”
count messages handled by MyAppRecalcSize

Synopsis: Keep a count of the methods called.

Notes: The args argument may be one of the following:

nothing start the method count or print the current count

‘reset’ reset the count to zero

‘stop’ stop the method count and remove it’s breakpoint
message handler
start the method count for a particular method

See Also: mwatch, showcalls.

■ memsize

Usage: memsize [<memory size>]

Examples:

“memsize”

“memsize 512”

Synopsis: Change the amount of memory that GEOS thinks that it has.



Notes:

- ◆ The <memory size> argument is the size to make the heap. If none is specified then the current memory size is returned.
- ◆ Memszie can only be run at startup, before the heap has been initialized. Use this right after an 'att -s'.
- ◆ Memszie accounts for the size of the stub.

■ methods

Usage: methods <class>
 methods <object>
 methods <flags>

Examples:

“methods -p” Print out methods defined for process

“methods ui::GenDocumentClass”
 Print out GenDocumentClass methods

“methods 3ffch:072fh”
 Print out methods for class at addr

“methods -a” Print methods of top class of app obj

Synopsis: Prints out the method table for the class specified, or if an object is passed, for the overall class of the object. Useful for getting a list of candidate locations to breakpoint.

■ model

Usage: model [<object>]

Examples:

“model” print model hierarchy from system object down

“model -i” print model hierarchy from implied grab down

“model ^l4e10h:20h”
 print model hierarchy from ^l4e10h:20h down.

Synopsis: Prints the model hierarchy below an object.

Notes:

- ◆ If no object is specified, the system object is used.
- ◆ The special object flags may be used to specify <object>. For a list of these flags, see pobject.
- ◆ Remember that the object you start from may have the model exclusive within its part of the hierarchy, but still not have the exclusive because something in a different part of the tree has it.
- ◆ The variable “printNamesInObjTrees” can be used to print out the actual app-defined labels for the objects, instead of the class, where available.
This variable defaults to false.

See Also: target, focus, mouse, keyboard, pobject.

■ modelobj

Usage: modelobj

Examples:

“modelobj” print model hierarchy from system object down

“pobj [modelobj]”

Do a **pobject** on the object with the model grab (the equivalent of a “pobj -m”).

Synopsis: Returns the object with the model grab.

See Also: target, focus, model, focusobj, targetobj.

■ mouse

Usage: mouse [<object>]

Examples:

“mouse” print mouse hierarchy from system object down

“mouse -i” print mouse hierarchy from implied grab down

“mouse ^l4e10h:20h”
print mouse hierarchy from ^l4e10h:20h down.

Synopsis: Prints the mouse hierarchy below an object.



Notes:

- ◆ If no object is specified, the system object is used.
- ◆ The special object flags may be used to specify <object>. For a list of these flags, see pobject.
- ◆ The variable “printNamesInObjTrees” can be used to print out the actual app-defined labels for the objects, instead of the class, where available.

This variable defaults to false.

See Also: target, focus, model, keyboard, pobject.

■ mouseobj

Usage: mouseobj

Examples:

“mouseobj” return object with mouse grab

“pobj [mouseobj]”

do a pobject on the object with the mouse grab (equivalent to “pobj -mg”).

Synopsis: Returns the object with the mouse grab.

See Also: target, focus, mouse, keyboard, keyboardobj.

■ mwatch

Usage: mwatch <msg>+
mwatch add <msg>+
mwatch list
mwatch clear

Examples:

“mwatch MSG_VIS_DRAW MSG_METAQUIT”
watch these messages

“mwatch add MSG_META_START_SELECT”
watch this message also

“mwatch” clear all message watches

Synopsis: Display all deliveries of a particular message.

Notes:

- ◆ The *msg* argument is which messages to watch. Those specified replace any messages watched before. If none are specified then any messages watched will be cleared.
- ◆ You may specify up to eight messages to be watched (fewer if you have other conditional breakpoints active). See *cbrk* for more information about conditional breakpoints.
- ◆ “mwatch clear” will clear all message watches.
- ◆ “mwatch add” will add the specified messages to the watch list.
- ◆ “mwatch list” will return a list of breakpoints that have been set by previous calls to mwatch.

See Also: objwatch, objbrk, objmessagebrk, procmessagebrk.

■ next

Usage: next

Examples:

“next” execute the next assembly instruction without entering it
“n”

Synopsis: Execute the patient by a single assembly instruction, skipping over any calls, repeated instructions, or software interrupts.

Notes:

- ◆ **next** does not protect against recursion, so when the breakpoint for the next instruction is hit, the frame of execution may be one lower.

See Also: step, istep.

■ noStructEnum

Usage: var noStructEnum [(0 | 1)]

Examples:

“var noStructEnum 1”
Don’t put “struct” or “enum” before the data type for variables that are structures or enumerated types.



Synopsis: Structure fields that are structures or enumerated types normally have “struct” or “enum” as part of their type description. This usually just clutters up the display, however, so this variable shuts off this prepending.

Notes: The default value of this variable is one.

See Also: print.

■ null

Usage: null <val>

Examples:

“null \$sym” Sees if the symbol token stored in \$sym is the empty string or “nil.”

Synopsis: Checks to see if a string is either empty or “nil,” special values returned by many commands when something isn’t found or doesn’t apply. Returns non-zero if <val> is either of these special values.

Notes: The notion of “nil” as a value comes from lisp.

See Also: index, range

■ objbrk

Usage: objbrk [<obj address>] [<message>]

Examples:

“objbrk ds:si MSG_VIS_DRAW”
break when a MSG_VIS_DRAW reaches the object

“objbrk -p” Break when any message is sent to the process object.

Synopsis: Break when a particular message reaches a particular object.

Notes:

- ◆ If you do not give a <message> argument after the <obj> argument, the machine will stop when any message is delivered to the object.
- ◆ <obj> is the address of the object to watch.
- ◆ The <objbrk> argument to “objbrk del” is the token/number returned when you set the breakpoint.

See Also: objwatch, objmessagebrk, mwatch.

■ obj-class

Usage: objclass <obj-addr>

Examples:

“var cs [obj-class ^lhx:si]”

Store the symbol token for the class of the object ^lhx:si in the variable \$cs.

Synopsis: Figures out the class of an object, coping with unrelocated object blocks and the like.

Notes:

- ◆ The value return is a symbol token, as one would pass to the “symbol” command. Using “symbol name” or “symbol fullname” you can obtain the actual class name.
- ◆ We decide whether to relocate the class pointer ourselves based on the LMF_RELOCATED bit in the *LMBH_flags* field of the object block’s header. There are times, e.g. during the call to MSG_META_RELOCATE for an object, when this bit doesn’t accurately reflect the state of the class pointer and we will return an error when we should not.

See Also: symbol.

■ objcount

Usage: objcount [-q] [-X] [-Y] [-b #] [-o #] [-p #]

Examples:

“objcount” count all objects

“objcount -p welcome”
count all objects owned by welcome

“objcount -o *desktop::DiskDrives”
count this one object

“objcount -b 0x3270”
count all objects in this block.

Synopsis: Count up instances of various objects on the heap.



Notes:

- ◆ The first argument specifies the options:
 - q** quiet operation - no progress output (not applicable with X, Y)
 - o #** check only object #
 - b #** check ONLY block #
 - p #** check only blocks for patient #
 - c #** check only objects of class #
 - C #** check only objects of top-level class #
 - X** show general verbose info
 - Y** show search verbose info
- ◆ Output fields:
 - direct** number of direct instances of this class
 - indirect** number if indirect instance of this class (i.e object's superclass is this class)
 - size** total size of instance data for this class (excludes instance data inherited from superclass)
- ◆ Status output:
 - . processing heap block
 - , processing matching object's top-level class
 - ; processing matching object's non-top-level class

See Also: hwalk, objwalk, lhwalk.

■ obj-foreach-class

Usage: obj-foreach-class <function> <object> [<args>]

Examples:

“obj-foreach-class foo-callback ^lhx:si”
 calls foo-callback with each class in turn to which the object
 ^lhx:si belongs.

Synopsis: Processes all the classes to which an object belongs, calling a callback procedure for each class symbol in turn.

Notes:

- ◆ <function> is called with the symbol for the current class as its first argument, <object> as its second, and the arguments that follow <object> as its third and subsequent arguments.
- ◆ <function> should return an empty string to continue up the class tree.
- ◆ obj-foreach-class returns whatever <function> returned, if it halted processing before the root of the class tree was reached. It returns the empty string if <function> never returned a non-empty result.

See Also: obj-class.

■ objmessagebrk

Usage: objmessagebrk [<address>]

Examples:

“objmessagebrk MyObj”
break whenever a message is sent to MyObj

“objmessagebrk”
stop intercepting messages

Synopsis: Break whenever a message is sent to a particular object via ObjMessage.

Notes:

- ◆ The <address> argument is the address to an object to watch for messages being sent to it. If no argument is specified then the watching is stopped.
- ◆ This breaks whenever a message is sent (before they get on the message queue. This enables one to track identical messages to an object which can be removed.

See Also: objwatch, mwatch, procmessagebrk, pobject.

■ objwalk

Usage: objwalk [<address>]

Examples: “objwalk”



Synopsis: Prints out information about an object block.

Notes:

- ◆ The <address> argument is the address of the block to print. The default is the block pointed at by DS.

See Also: lhwalk, pobject

■ objwatch

Usage: objwatch [<address>]

Examples:

“objwatch ds:si”
watch the messages which reach the object at DS:SI

“objwatch MyObject”
watch the messages which reach MyObject

“objwatch”
Watch the messages which reach the process object.

Synopsis: Display message calls that have reached a particular object.

Notes:

- ◆ The <address> argument is the address of the object to watch.
- ◆ This returns the token of the breakpoint being used to watch message deliveries to the object. Use the “brk” command to enable, disable, or turn off the watching of the object.

See Also: brk, mwatch, objmessagebrk, procmessagebrk, pobject.

■ omfq

Usage: omfq <message> <object> <args>*

Examples:

“omfq MSG_META_QUIT *HelloApp”
Sends MSG_META_QUIT to the ***HelloApp** object.

Synopsis: Forces a message for an object onto its event queue.

Notes:

- ◆ This command calls `ObjMessage`, passing it `di=mask MF_FORCE_QUEUE`.
- ◆ `<args>` is the set of additional parameters to pass to `ObjMessage`. It consists of `<variable/register> <value>` pairs, which are passed to the “assign” command. As a special case, if the variable is “push”, the value (a word) is pushed onto the stack and is popped when the message has been queued.
- ◆ The registers active before you issued this command are always restored, regardless of whether the call to **ObjMessage** completes successfully. This is in contrast to the “call” command, which leaves you where the machine stopped with the previous state lost.

See Also: `call`.

■ pappcache

Usage: `pappcache`

Examples:

“pappcache” Print out current state of the app-cache

Synopsis: Prints out the current state of the system application cache, for systems operating in transparent launch mode.

Notes: Specifically, this command prints out:

- ◆ Applications in the cache (First choice for detaching)
- ◆ Top full-screen App (Not detached except by another full screen app)
- ◆ Desk accessories (detached only as last resort)
- ◆ Application geodes in the process of detaching

■ patch

Usage: `patch [<addr>]`
`patch del <addr>*`

Synopsis: Patch assists in creating breakpoints that invisibly make small changes to code. This can help the programmer find several bugs without remaking and redownloading.



Notes:

- ◆ If you give no <addr> when creating a patch, the patch will be placed at the most-recently accessed address, as set by the command that most-recently accessed memory (e.g. bytes, words, listi, imem, etc.)
- ◆ When creating a patch, you are prompted for its contents, each line of which comes from the following command set: (see Table 4-2)

Table 4-2 Patch Command Set

Form	Meaning	Example
<reg> = <value>	assign value to reg	ax = bx dl = 5
push <reg> <value>	push value	push ax push 45
pop <reg> <value>	pop value	pop ax pop 45
pop	pop nothing (sp=sp+2)	pop
jmp <address>	change ip	jmp UI_Attach+45
scall <address> <regs>	call routine (save)	scall MemLock ax = 3
mcall<address> <regs>	call routine (modify)	mcall MemLock ax = 3
xchg <reg> <reg>	swap two registers	xchg ax bx
set <flag>	set condition flag	set CF set ZF
reset <flag>	reset condition flag	reset CF reset ZF
if <flag>	if flag set then ...	if CF
if !<flag>	if flag reset then ...	if !ZF
if <expr>	if expr then ...	if foo == 4
else		
endif		
ret	make function return	ret
\$	terminate input	
a	abort	
<other>	tcl command	echo \$foo

<flag> is taken from the set TF, IF, DF, OF, SF, ZF, PF, AF, CF and must be in upper-case.

The “scall” command has no effect on the current registers (not even for purposes of return values), while the “mcall” command changes whatever registers the function called modifies. See the “call” documentation for the format of <regs>.

■ patchin

Patchin undoes the work of patchout.



■ patchout

This command causes a RET to be placed at the start of a routine.

■ patient This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ patient-default

Usage: patient-default [<geode-name>]

Examples:

“patient-default hello2”
 Makes “hello2” the default patient.

“patient-default”
 Prints the names of the current default patient.

Synopsis: Specifies the default patient. The **send** and **run** commands will use this as the default patient to operate on if none is passed to them.

■ pbitmap

Usage: pbitmap <address>

Synopsis: Print a bitmap graphically.

Notes:

- ◆ The address argument is the address of the Bitmap or CBitmap structure.
- ◆ Color bitmaps are printed with a letter representing the color as well. The letters are index from the string (kbgcrvnAaBGCRVYW).

■ pbody

Usage: pbody [<address>]

Examples:

“pbody” prints the GrObjBody given a GrObj block at DS.

“pbody ^hbx”
 Prints the GrObjBody given a GrObj block whose handle is BX.

Synopsis: Finds the GrObjBody—prints its OD and its instance data.



Notes: If no arguments are given, then DS is assumed to point to an object block containing GrObjects.

■ parray

Usage: parray [-eth] [<address>]

Examples:

“parray” Print the chunk array at *DS:SI (header only)

“parray es:di”
Print the chunk array at ES:DI (header only)

“parray -e” Print the chunk array at *DS:SI and print the elements in the array

“parray -tMyStruct”
Print the chunk array at *DS:SI and print the elements where the elements are of type MyStruct

“parray -tMyStruct -TMyExtraStruct”
Like above, but data after MyStruct is printed as an array of MyExtraStruct structures.

“parray -e3”
Print the chunk array at *DS:SI and print the third element

“parray -hMyHeader”
Print the chunk array at *DS:SI (header only) where the header is of type MyHeader

Synopsis: Print information about a chunk array.

Notes:

- ◆ The flags argument can be any combination of the flags “e”, “t”, and “h”. The “e” flag prints all elements. If followed by a number (e.g. “-e3”), then only the third element will be printed.
- ◆ The ‘t’ flag specifies the elements’ type. It should be followed immediately by the element type. You can also use “-tgstring” if the elements are GString Elements.
- ◆ The ‘h’ flag specifies the header type. It should be followed immediately by the element type.
- ◆ The ‘l’ flag specifies how many elements to print. It can be used in conjunction with the ‘e’ flag to print a range of element numbers.
- ◆ The ‘H’ flag suppresses printing of the header.
- ◆ All flags are optional and may be combined.
- ◆ The address argument is the address of the chunk array. If not specified then *ds:si is used.

■ pcbitmap

Usage: pcbitmap <address> <width> <height> [<no space flag>]

Examples:

```
“pcbitmap *ds:si 64 64 t”  
    print the bitmap without spaces
```

Synopsis: Print out a one-bit deep packbits-compacted bitmap.

Notes:

- ◆ The <address> argument is the address to the bitmap data.
- ◆ The <width> argument is the width of the bitmap in pixels.
- ◆ The <height> argument is the height of the bitmap in pixels.
- ◆ The <no space flag> argument removes the space normally printed between the pixels. Anything (like “t”) will activate the flag.

See Also: pncbitmap.



■ pcelldata

Usage: pcelldata [<addr>]

Examples:

“pcelldata *es:di”
Print cell data for cell at *es:di.

Synopsis: Prints data for a spreadsheet data.

Notes: If no address is given, *es:di is used.

See Also: content, pcelldeps.

■ pcelldeps

Usage: pcelldeps <filehan> [<addr>]

Examples:

“pcelldeps 4be0h *es:di”
print dependencies of cell in file 4be0h.

Synopsis: Prints dependencies for a cell in the spreadsheet.

Notes: If no address is given, *es:di is used.

See Also: content, pcelldata.

■ pclass

Usage: pclass [<address>]

Examples:

“pclass” prints the class of *DS:SI
“pclass ^l4ac0h:001eh”
Print the class of the object at the given address.

Synopsis: Print the object's class.

Notes:

- ◆ The <address> argument is the address of the object to find the class of. This defaults to *DS:SI.

■ pdb

Produces useful information about a DBase block. For now, only information about the map block of the DBase file is produced. First arg H is the SIF_FILE or SIG_VM handle's ID. Second arg B is the VM block handle for which information is desired.

■ pdisk

Usage: pdisk <disk-handle>

Examples:

“pdisk 00a2h”

Prints information about the disk whose handle is 00a2h.

Synopsis: Prints out information about a registered disk, given its handle.

Notes: The Flags column is a string of single-character flags with the following meanings:

- w** The disk is writable.
- V** The disk is always valid, i.e. it's not removable.
- S** The disk is stale. This is set if the drive for the disk has been deleted.
- u** The disk is unnamed, so the system has made up a name for it.

See Also: diskwalk, drivewalk.

■ pdrive

Usage: pdrive <drive-handle>
pdrive <drive-name>
pdrive <drive-number>

Examples:

- “pdrive si” Print a description of the drive whose handle is in SI
- “pdrive al” Print a description of the drive whose number is in AL
- “pdrive C” Print a description of drive C



Synopsis: Provides the same information as “drivewalk,” but for a single drive, given the offset to its DriveStatusEntry structure in the FSInfoResource.

Notes: This is intended for use by implementors of IFS drivers, as no one else is likely to ever see a drive handle.

See Also: drivewalk.

■ penum

Usage: penum <type> <value>

Examples:

“penum FatalErrors 0”
print the first FatalErrors enumeration

Synopsis: Print an enumeration constant given a numerical value.

Notes:

- ◆ The <type> argument is the type of the enumeration.
- ◆ The <value> argument is the value of the enumeration in a numerical format.

See Also: print, precord.

■ pevent

Usage: pevent <handle>

Examples:

“pevent 39a0h”
Print event with handle.

Synopsis: Print an event given its handle.

See Also: elist, eqlist, eqfind, erfind.

■ pflags

Usage: pflags

Synopsis: Prints the current machine flags (carry, zero, etc.).

See Also: setcc, getcc.

■ pfont

Usage: pfont [-c] [<address>]

Examples:

“pfont” print bitmaps of the characters of the font in BX.

“pfont -c ^h1fd0h”
list the characters in the font at ^h1fd0h.

Synopsis: Print all the bitmaps of the characters in a font.

Notes:

- ◆ The “-c” flag causes pfont to list which characters are in the font and any special status (i.e. NOT BUILT).
- ◆ The <address> argument is the address of the font. If none is specified then ^hbx is used.

See Also: fonts, pusage, pfontinfo.

■ pfontinfo

Usage: pfontinfo

Examples: “pfontinfo FID_BERKELEY”

Synopsis: Prints font header information for a font. Also lists all sizes built.

Notes:

- ◆ The argument must be supplied. If not known, use ‘fonts -u’ to list all the fonts with their IDs.

See Also: fonts, pfont.



■ pgen

Usage: pgen <element> [<object>]

Examples:

“pgen GI_states @65”
print the states of object 65

“pgen GI_visMoniker”
print the moniker of the object at *DS:SI

“pgen GI_states -i”
print the states of the object at the implied grab

Synopsis: Print an element of the generic instance data.

Notes:

- ◆ The <element> argument specifies which element in the object to print
- ◆ The <object> argument is the address to the object to print out. It defaults to *DS:SI and is optional. The ‘-i’ flag for an implied grab may be used.

See Also: gentree, gup, pobject, piv, pvis.

■ pgs

Usage: pgs <address>

Examples:

“pgs” List the graphics string at DS:SI

“pgs ^hdi” Lift the graphics string whose handle is in DI, starting at the current position.

“pgs -s ^hdi” List the graphics string whose handle is in DI, starting at the beginning of the graphics string.

“pgs -l3 ^hdi”
List three elements of the graphics string whose handle is in DI, starting at the current position.

Synopsis: List the contents of a graphics string.

Notes:

- ◆ The <address> argument is the address of a graphics string. If none is specified then DS:SI is used as a pointer to a graphics string.
- ◆ The passed address may also be the base of a gstate (e.g. “^hdi”). In this case, the gstring that is associated with the gstate will be printed.
- ◆ The -s option can be used to specify that the gstring should be listed from the beginning of the string. By default, gstrings will be listed starting at the current position.
- ◆ The -g option can be used to specify that the passed address is the address of a GrObj (GStringClass) object— the gstring for that object will be listed.

See Also: pbitmap.

■ phandle

Usage: phandle <handle ID>

Examples:

```
“phandle 1a8ch”  
    print the handle 1a8ch
```

Synopsis: Print out a handle.

Notes:

- ◆ The <handle ID> argument is just the handle number. Make sure that the proper radix is used.
- ◆ The size is in paragraphs.

See Also: hwalk, lhwalk.



■ pharray

Usage: pharray [<flags>] [<vmfile> <dirblk>]

Examples:

“pharray” Print the huge array at ^vbx:di (header only)

“pharray dx cx”
Print the huge array at ^vdx:cx (header only)

“pharray -e” Print the huge array at ^vbx:di and print the elements in the array

“pharray -tMyStruct”
Print the huge array at ^vbx:di and print the elements where the elements are of type MyStruct

“pharray -e3”
Print the huge array at ^vbx:di and print the third element

“pharray -h” Print the header of the HugeArray at ^vbx:di, using the default header type (HugeArrayDirectory).

“pharray -hMyHeader”
Print the huge array at ^vbx:di (header only) where the header is of type MyHeader

“pharray -d” Print the directory elements of a HugeArray

“pharray -e5 -l8”
Print 8 HugeArray elements starting with number 5

Synopsis: Print information about a huge array.

Notes:

- ◆ The flags argument can be any combination of the flags ‘e’, ‘t’, and ‘h’. The ‘e’ flag prints all elements. If followed by a number “-e3”, then only the third element is printed.

The ‘t’ flag specifies the elements’ type. It should be followed immediately by the element type. You can also use “gstring”, in which case the elements will be interpreted as GString Elements.

The ‘h’ flag specifies the header type. It should be followed immediately by the element type. If no options are specified, then “-hHugeArrayDirectory” is used. If any other options are specified, then

the printing of the header is disabled. So, for example, if you want both the header and the third element, use “-h -e3”.

The ‘d’ flag specifies that the HugeArray directory entries should be printed out.

The ‘l’ flag specifies how many elements to print.

The ‘s’ flag requests that a summary table be printed.

All flags are optional and may be combined.

- ◆ The address arguments are the VM file handle and the VM block handle for the directory block. If nothing is specified, then bx:di is used

■ pini

Usage: pini [<category>]

Examples:

“pini Lights Out”

Print out the contents of the Lights Out category in each .ini file

“pini”

Print out each currently loaded .ini file.

Synopsis: Provides you with the contents of the .ini files being used by the current GEOS session.

Notes: <category> may contain spaces and other such fun things. In fact, if you attempt to quote the argument (e.g. “pini {Lights Out}”), this will not find the category.

■ pinst

Usage: pinst [<address>]

Examples:

“pinst” print the last master level of instance data of the object at *DS:SI

“pinst *MyObject”

print the last master level of instance data of MyObject.

“pinst -i”

print the last master level of the windowed object at the mouse pointer.



Synopsis: Print out all the instance data to the last level of the object.

Notes:

- ◆ The <address> argument is the address of the object to examine. If not specified then **pinst** will use a default address. If you are debugging a C method, then the **oself** value will be used. Otherwise, ***DS:SI** is assumed to be an object.
- ◆ This command is useful for classes you've created and you are not interested in the data in the master levels which **pobject** would display.
- ◆ The following special values are accepted for <address>:
 - a** the current patient's application object
 - i** the current "implied grab": the windowed object over which the mouse is currently located.
 - f** the leaf of the keyboard-focus hierarchy
 - t** the leaf of the target hierarchy
 - m** the leaf of the model hierarchy
 - c** the content for the view over which the mouse is currently located
 - kg** the leaf of the keyboard-grab hierarchy
 - mg** the leaf of the mouse-grab hierarchy
- ◆ **pinst** prints out the same information as "**pobj l**".

See Also: **pobject**, **piv**.

■ **piv**

Usage: **piv** <master> <iv> [<address>]

Examples:

```
"piv Vis VCNI_viewHeight"
    print Vis.VCNI_viewHeight at *DS:SI
```

Synopsis: This prints out the value of the instance variable specified.

Notes:

- ◆ The <master> argument expects the name of a master level. The name may be found using **pobject** to print the levels, and then using the name that appears after “master part:” and before the “_offset”.
- ◆ The <iv> argument expects the name of the instance variable to print.
- ◆ The <address> argument is the address of the object to examine. If not specified then *DS:SI assumed to be an object.
- ◆ This command is useful for when you know what instance variable you want to see but don't want to wade through a whole pobject command.

See Also: pobject, pinst.

Synopsis: Prints a keyboard map in assembly-language format.

■ **plines**

Usage: plines <start> [<obj-address>]

Examples:

“plines 12” Print lines starting at line 12.

“plines 12 ^l6340h:0020h”
 Print lines starting at line 12 of object at given address.

Synopsis: Print information about the lines in a text object.

Notes: The printed line-starts are *not* correct.

See Also: ptext.

■ **plist**

Prints out a list of structures stored in an lmem chunk. It takes two arguments, the structure type that makes up the list, and the lmem handle of the chunk. e.g. plist FontsInUseEntry ds:di



■ pncbitmap

Usage: pncbitmap <address> <width> <height> [<no space flag>]

Examples:

“pncbitmap *ds:si 64 64 t”
print the bitmap without spaces

Synopsis: Print out a one-bitdeep noncompacted bitmap.

Notes:

- ◆ The <address> argument is the address to the bitmap data.
- ◆ The <width> argument is the width of the bitmap in pixels.
- ◆ The <height> argument is the height of the bitmap in pixels.
- ◆ The <no space flag> argument removes the space normally printed between the pixels. Anything (like ‘t’) will activate the flag.

See Also: pcbitmap.

■ pnormal

Usage: pnormal [-v]

Examples:

“pnormal -v”
Print out verbose information about the current normal transfer item.

Synopsis: Prints out information about the current “normal” transfer item on the clipboard.

Notes: If you give the “-v” flag, this will print out the contents of the different transfer formats, rather than just an indication of their types.

See Also: pquick, print-clipboard-item.

■ pobjarray

Usage: pobjarray [<address>]

Examples:

“pobjarray” Print the array of ODs at *ds:si.

Synopsis: Print out an array of objects.

See Also: pbody.

■ pobject

Usage: pobject [<address>] [<detail>]

Examples:

“pobj” print the object at *ds:si from Gen down if Gen is one of its master levels; else, print all levels

“pobj *MyGenObject”
print MyGenObject from Gen down

“pobj Gen” print the Gen level for the object at *ds:si

“pobj last” print the last master level for the object at *ds:si

“pobj *MyObject”
all print all levels of MyObject

“pobj -i sketch”
print the master level headings of the windowed object at the mouse pointer

“pobj *MyObject FI_foo”
print the FI_foo instance variable for MyObject

“pobj HINT_FOO”
print the HINT_FOO variable data entry for the object at *ds:si

“pobj v” print the variable data for the object at *ds:si

Synopsis: Print all or part of an object’s instance and variable data.



Notes:

- ◆ The <address> argument is the address of the object to examine. If not specified then *oself* is used, unless the current function is written in assembly, in which case **DS:SI* .
- ◆ The following flag values are accepted in lieu of an address:
 - a the current patient's application object
 - i the current "implied grab": the windowed object over which the mouse is currently located.
 - f the leaf of the keyboard-focus hierarchy
 - t the leaf of the target hierarchy
 - m the leaf of the model hierarchy
 - c the content for the view over which the mouse is currently located
 - kg the leaf of the keyboard-grab hierarchy
 - mg the leaf of the mouse-grab hierarchy
- ◆ The *detail* argument specifies what information should be printed out about the object. If none is specified, all levels of the object from the Gen level down will be printed if Gen is one of the object's master levels; else, the whole object will be printed.
- ◆ The following values are accepted for *detail*:
 - all** (or **a**)
all master levels
 - last** (or **l**)
last master level only
 - sketch** (or **s**)
master level headings only
 - vardata** (or **v**)
vardata only
 - a master level name
 - an instance variable name
 - a variable data entry name

See Also: pinst, piv, pvardata.

■ pobjmon

Usage: pobjmon [<address>] [<text only>]

Examples:

“pobjmon” print the VisMoniker from the gentree object at *DS:SI

Notes:

- ◆ The <address> argument is the address of an object with a VisMoniker. If none is specified then *DS:SI is used.
- ◆ The <text only> argument returns a shortened description of the structure. To set it use something other than ‘0’ for the second argument.
- ◆ The special object flags may be used to specify <object>. For a list of these flags, see pobject.

See Also: pvismon, pobject, vistree, gup, gentree, impliedgrab, systemobj.

■ pod

Usage: pod <address>

Examples:

“pod ds:si”

Synopsis: Print in output descriptor format (^l<handle>:<chunk>) the address passed.

Notes: The address argument is the address of an object.

■ ppath

Usage: ppath (current | docClip | winClip) [<gstate>]

Examples:

“ppath” print the current path of the GState in ^hdi

“ppath docClip ^hgstate”
 print the document clip path of the GState with handle *gstate*.

“ppath winClip ds”
 print the window clip path of the GState in the DS register.



Synopsis: Print the structure of a path.

Notes: Unique abbreviations for the path to be printed are allowed.

■ pquick

Usage: pquick [-v]

Examples:

“pquick -v” Print out verbose information about the current quick transfer item.

Synopsis: Prints out information about the current “quick” transfer item on the clipboard.

Notes: If you give the “-v” flag, this will print out the contents of the different transfer formats, rather than just an indication of their types.

See Also: pnormal, print-clipboard-item.

■ precord

Usage: precord <type> <value> [<silent>]

Examples:

“precord GSControl c0h”
print the **GSControl** record with the top two bits set

Synopsis: Print a record using a certain value.

Notes:

- ◆ The <type> argument is the type of the record.
- ◆ The <value> argument is the value of the record.
- ◆ The <silent> argument will suppress the text indicating the record type and value. This is done by passing a non zero value like ‘1’. This is useful when precord is used by other functions.

See Also: print, penum.

■ preg

Usage: preg [-g] <addr>

Examples:

“preg *es:W_appReg”

Prints the application-defined clipping region for the window pointed to by es.

“preg -g ds:si”

Prints a “graphical” representation of the region beginning at ds:si

Synopsis: Decodes a graphics GEOS region and prints it out, either numerically, or as a series of x's and spaces.

Notes:

- ◆ This command can deal with parameterized regions. When printing a parameterized region with the -g flag, the region is printed as if it were unparameterized, with the offsets from the various PARAM constants used as the coordinates.
- ◆ If no address is given, this will use the last-accessed address (as the “bytes” and “words” commands do). It sets the last-accessed address, for other commands to use, to the first byte after the region definition.

■ print

Usage: print <expression>

Examples:

“print 56h” print the constant 56h in various formats

“print ax - 10”
print ax less 10 decimal

“print ^l31a0h:001eh”
print the absolute address of the pointer

Synopsis: Print the value of an expression.



Notes:

- ◆ The <expression> argument is usually an address that has a type or that is given a type by casting and may span multiple arguments. The contents of memory of the given type at that address is what's printed. If the expression has no type, its offset part is printed in both hex and decimal. This is used for printing registers, for example.

- ◆ The first argument may contain the following flags (which start with '-'):

x	integers (bytes, words, dwords if dwordIsPtr false) printed in hex
d	integers printed in decimal
o	integers printed in octal c bytes printed as characters (byte arrays printed as strings, byte variables/fields printed as character followed by integer equivalent)
C	bytes treated as integers
a	align structure fields
A	Don't align structure fields
p	dwords are far pointers
P	dwords aren't far pointers
r	parse regions
R	don't try to parse regions

- ◆ These flags operate on the following Tcl variables:

intFormat A printf format string for integers.

bytesAsChar
Treat bytes as characters if non-zero.

alignFields
Align structure fields if non-zero.

dwordIsPtr DWords are far pointers if non-zero.

noStructEnum
If non-zero, doesn't print the "struct", "enum" or "record" before

the name of a structured/enumerated type -- just gives the type name.

printRegions

If non-zero, prints what a Region points to (bounds and so on).

condenseSpecial

If non-zero, condense special structures (Rectangles, OutputDescriptors, ObjectDescriptors, TMatrixes and all fixed-point numbers) to one line.

◆ This does not print enumerations. Use penum for that.

See Also: precord, penum, _print.

■ **print-cell**

Usage: print-cell [row column <cfp ds:si>]

Examples:

“print-cell 1 1”
print the cell <1,1>
“print-cell 1 2 *ds:si”
print the cell <1,2> given *DS:SI

Synopsis: Print information about a cell

See Also: print-row, print-row-block, print-cell-params, print-column-element

■ **print-cell-params**

Usage: print-cell-params [<address>]

Examples:

“print-cell-params”
print the **CellFunctionParameters** at ds:si.
“print-cell-params ds:bx”
print the **CellFunctionParameters** at ds:bx.

Synopsis: Print a **CellFunctionParameters** block.

See Also: print-row, print-column-element, print-row-block, print-cell.



■ print-clipboard-item

Usage: print-clipboard-item [-v] <vmfile> <vmblock>
 print-clipboard-item [-v] <memhandle>
 print-clipboard-item [-v] <addr>

Examples:

“print-clipboard-item bx”
 Print out info about the transfer item whose memory handle is
 in the BX register.

Synopsis: Prints out information about a transfer item.

Notes:

- ◆ If you give the “-v” flag, this will print out the contents of the different transfer formats, rather than just an indication of their types.
- ◆ The -v flag will not work unless the transfer item is in a VM file.

See Also: pnormal, pquick.

■ print-column-element

Usage: print-column-element [<address>]

Examples:

“print-column-element”
 Print the **ColumnArrayElement** at ds:si.

“print-column-element ds:bx” print the **ColumnArrayElement** at ds:bx

Synopsis: Print a single **ColumnArrayElement** at a given address.

■ print-db-group

Usage: print-db-group file group

Examples:

“print-db-group ax bx”
 print the group at bx/ax.

Synopsis: Print information about a dbase group block.

See Also: print-db-item.

■ print-db-item

Usage: print-db-item file group item

Examples:

“print-db-item bx ax di”
print the item at bx/ax/di

Synopsis: Print information about a single dbase item

See Also: print-db-group

■ print-eval-dep-list

Usage: print-eval-dep-list [<addr>]

Examples:

“print-eval-dep-list es:0”
Print dependency list at ES:0.

Synopsis: Prints a dependency list used for evaluation.

See Also: content, pcelldeps.

■ printNamesInObjTrees

Usage: var printNamesInObjTrees (0 | 1)

Examples:

“var printNamesInObjTrees 1”
Sets “gentree,” “vistree,” etc. commands to print object names
(where available).

Synopsis: Determines whether object names are printed (where available) rather than class names when using the following commands: vistree, gentree, focus, target, model, mouse, keyboard.

Notes: The default value for this variable is zero.

See Also: gentree, vistree, focus, target, model, mouse, keyboard.



■ print-obj-and-method

Usage: print-obj-and-method <handle> <chunk> <message> [<cx> [<dx> [<bp> [<class>]]]]

Examples:

“print-obj-and-method [read-reg bx] [read-reg si]”
Prints a description of the object ^l_{bx}:si with the value stored and a hex representation.

“print-obj-and-method \$h \$c \$m [read-reg cx] [read-reg dx] [read-reg bp]”
Prints a description of the object ^l\$_h:\$_c and the name of the message whose number is in \$_c. This is followed by the three words of data in cx, dx, and bp.

Synopsis: Prints a nicely formatted representation of an object, with option message, register data, label, hex address, & carriage return. The class indication may also be overridden.

Notes:

- ◆ You may specify anywhere from 0 to 5 arguments after the message number. These are interpreted as the value of the message, the registers CX, DX and BP, and the symbol token of the class to print, respectively.
- ◆ All arguments must be integers, as this is expected to be called by another procedure, not by the user, so the extra time required to call **getvalue** would normally be wasted. (The user should call **pobj**, **gup**, or other such functions for this sort of print out.)

See Also: mwatch, map-method, objwatch.

■ printRegions

Usage: var printRegions [(0 | 1)]

Examples:

“var printRegions 1”
If a structure contains a pointer to a region, “print” will attempt to determine its bounding box.

Synopsis: Controls whether “print” parses regions to find their bounding rectangle.

Notes: The default value for this variable is one.

See Also: print, condenseSpecial.

■ print-row

Usage: print-row [<address *DS:SI>]

Examples:

“print-row” print the row at *DS:SI

“print-row ds:si”
print the row at DS:SI

Synopsis: Print a single row in the cell file given a pointer to the row.

See Also: print-column-element print-cell-params print-row-block print-cell

■ print-row-block

Usage: print-row-block [<address ds>]

Examples:

“print-row-block”
print the row-block at DS:0

“print-row-block es”
print the row-block at ES:0

Synopsis: Print a row-block.

See Also: print-row, print-cell-params, print-column-element, print-cell.



■ printStop

Synopsis: This variable controls how the current machine state is printed each time the machine comes to a complete stop. Possible values:

asm	Print the current assembly-language instruction, complete with the values for the instruction operands.
src	Print the current source line, if it's available. If the source line is not available, the current assembly-language instruction is displayed as above.
why	Print only the reason for the stopping, not the current machine state. "asm" and "src" modes also print this.
nil	Don't print anything.

■ **proc** This is a Tcl primitive. See "Tool Command Language," Chapter 5.

■ procmessagebrk

Usage: procmessagebrk [<handle>]

Examples:

```
"procmessagebrk MyObj"
    break whenever a message is sent to MyObj

"procmessagebrk"
    stop intercepting messages
```

Synopsis: Break whenever a message is *sent* to a particular process via ObjMessage.

Notes:

- ◆ The <handle> argument is the handle to a process to watch for messages being sent to it. If no argument is specified then the watching is stopped. The process' handle may be found by typing "ps -p". The process's handle is the number before the process's name.
- ◆ This command breaks whenever a message is sent (before they get on the message queue. This enables one to track identical messages to a process which can be removed.

See Also: objwatch, mwatch, objmessagebrk, pobject.

■ **protect** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ **ps**

Usage: ps [<flags>]

Examples:

“ps -t” list all threads in GEOS.

Synopsis: Print out GEOS’ system status.

Notes: The flags argument may be one of the following:

-t Prints out info on all threads. May be followed by a list of patients whose threads are to be displayed.

-p Prints out info on all patients. May be followed by a list of patients to be displayed.

-h Prints out info on all handles. May be followed by a list of patients whose handles are to be displayed.

The default is ‘-p’.

See Also: switch, sym-default.

■ **pscope**

Usage: pscope [<scope-name> [<sym-class>]]

Examples:

“pscope WinOpen”

Prints out all the local labels, variables, and arguments within the **WinOpen()** procedure

Synopsis: This prints out all the symbols contained in a particular scope.

Notes:

◆ This can be useful when you want to know just the fields in a structure, and not the fields within those fields, or if you know the segment in which a variable lies, but not its name. Admittedly, this could be overkill.

◆ *sym-class* can be a list of symbol classes to restrict the output. For example, “pscope Filemisc proc” will print out all the procedures within the Filemisc resource.



See Also: whatis, locals.

■ psize

Usage: psize <structure>

Examples: “psize FontsInUseEntry”

Synopsis: Print the size of the passed structure.

■ pssheet

Usage: pssheet [-isSfrcvd] <address>

Examples:

“pssheet -s ^l3ce0h:001eh”
 print style attributes.

“pssheet -f -i 94e5h:0057h”
 print file info from instance data.

Synopsis: Prints out information about a spreadsheet object.

Notes:

- ◆ If you are in the middle of debugging a spreadsheet routine and have a pointer to the Spreadsheet instance, the “-i” flag can be used to specify the object using that pointer.
- ◆ If you simply have the OD of the spreadsheet object, use that.
- ◆ Alternatively, you can do:

 pssheet <flags> [targetobj]

See Also: content, targetobj.

■ psup

Usage: psup [<object>]

Examples:

“psup” print superclasses of object at *ds:si.
“psup -i” print superclasses of object under mouse.
“psup ^l4e10h:20h”
 print superclasses of object at ^l4e10h:20h.

Synopsis: Prints superclasses of an object.

Notes: If no object is specified, *ds:si is used.

See Also: is-obj-in-class.

■ ptext

Usage: ptext [-lsrtcegR] <address>

Synopsis: Prints out a text object

Notes: The flag may be one of the following:

-c Print out the characters (the default).
-e print out elements in addition to runs.
-l print out line and field structures.
-s print out char attr structures.
-r print out para attr structures.
-g print out graphics structures.
-t print out type structures.
-R print out region structures.



■ pthread

Usage: pthread <id>

Examples:

“pthread 16c0h”

Prints information about the thread whose handle is 16c0h.

Synopsis: Provides various useful pieces of information about a particular thread including its current priority and its current registers.

Notes: <id> is the thread's handle ID, as obtained with the “ps -t” or “threadstat” command.

See Also: os, threadstat.

■ ptimer

Usage: ptimer <handle>

Examples:

“ptimer bx” Print out information about the timer whose handle is in the BX register.

Synopsis: Prints out information about a timer registered with the system: when it will fire, what it will do when it fires, etc.

Notes: <handle> may be a variable, register, or constant.

See Also: twalk, phandle.

■ ptrans

Usage: ptrans [<flags>] [<address>]

Examples:

“ptrans” print the normal transform for the object at *ds:si.

“ptrans -s” print the sprite transform for the GrObj object at *ds:si.

“ptrans ^lbx:cx”
print the normal transform for the object whose OD is ^lbx:cx.

Synopsis: Prints the **ObjectTransform** data structure as specified.

Notes:

- ◆ The -s flag can be used to print the “sprite” transform (the “sprite” is the shape’s outline which is drawn to give feedback to the user when said user is moving/rotating/etc. the GrObj).
- ◆ <address> defaults to *ds:si

See Also: pobject.

■ **ptreg**

Usage: ptreg <start> [<obj-addr>]

Examples:

“ptreg 12” Print lines for region 12
“ptreg 12 ^lcx:dx”
 Print lines for region 12 of object ^lcx:dx

Synopsis: Print information about the lines in a region.

See Also: ptext.

■ **pusage**

Usage: pusage [<address>]

Examples:

“pusage” print the usage of characters in the font

Synopsis: List the characters in a font and when they were last used.

Notes:

- ◆ The <address> argument is the address of a font. If none is given then ^hbx is used.

See Also: fonts, pfont, pfontinfo, plist.



■ pvardata

Usage: pvardata [<entry>]

Examples:

“pvardata ds:si”
Prints vardata of object at *ds:si

“pvardata -i”
Prints vardata of object with implied grab.

Notes:

- ◆ The address argument is the address of an object with variable data. The default is *ds:si.

■ pvardentry

Usage: pvardentry <address> <object>

Examples:

“pvardentry ds:bx *ds:si”

Notes:

- ◆ The address argument is the address of a variable data entry in an object’s variable data storage area. The default is ds:bx.
- ◆ The <object> argument is required to determine the name of the tag for the entry, as well as the type of data stored with it.

■ pvis

Usage: pvis <element> [<object>]

Examples:

“pvis VI_bounds @65”
print the bounds of object 65

“pvis VI_optFlags”
print the flags of the object at *DS:SI

“pvis VI_attr -i”
print the attributes of the object at the implied grab

Synopsis: Print an element of the visual instance data.

Notes:

- ◆ The <element> argument specifies which element in the object to print
- ◆ The <object> argument is the address to the object to print out. It defaults to *DS:SI and is optional. The '-i' flag for an implied grab may be used.

See Also: vistree, vup, pobject, piv, pgen.

■ pvismon

Usage: pvismon [<address>] [<text only>]

Examples:

"pvismon" print the moniker at *DS:SI
"pvismon -i 1" print a short description of the implied grab object.

Synopsis: Print a visual moniker structure at an absolute address.

Notes:

- ◆ The <address> argument is the address to an object in the visual tree. This defaults to *DS:SI. The '-i' flag for an implied grab may be used.
- ◆ The <text only> argument returns a shortened description of the structure. Pass a non-zero value to turn on this flag.
- ◆ The special object flags may be used to specify <object>. For a list of these flags, see pobject.

See Also: pobjmon, pobject, vistree, gup, gentree, impliedgrab, systemobj.

■ pvmb

Synopsis: Prints out the VMBlockHandle for a VM block given the file handle *H* and the VM block handle *B*.



■ pvmt

Usage: pvmt [-p] [-a] [-s] [-c] (<handle> | <segment>)

Examples:

“pvmt bx” Print out all used blocks for the open VM file whose file handle is in BX.

“pvmt -as ds” Print out all blocks for the open VM file the segment of whose header block is in DS.

Synopsis: Prints out a map of the VM block handles for a VM file.

Notes:

- ◆ The -p flag will only print out blocks that have the Preserve flag set. Useful for examining object blocks in GeoCalc files, for example.
- ◆ The -a flag causes pvmt to print out all block handles, not just those that have been allocated. The other two types of block handles are “assigned” (meaning they’re available for use, but currently are tracking unused space in the file) and “unassigned” (they’re available for use).
- ◆ The -s indicates the final argument is a segment, not a file handle. This is used only if you’re inside the VM subsystem of the kernel.
- ◆ The -c flag requests a count of the different types of blocks at the end of the printout.
- ◆ The blocks are printed in a table with the following columns:

han	VM block handle (in hex)
flags	D if the block is dirty, C if the block is clean, - if the block is non-resident, L if the block is LMem, B if the block has a backup,

P if the preserve handle bit is set for the block,
! if the block is locked

memhan Associated memory handle. Followed by “(d)” if the memory for the block was discarded but the handle retained. Followed by (s) if the memory has been swapped out.

block type The type of block:

VMBT_USED a normal in-use block,

VMBT_DUP an in-use block that has been backed up or allocated since the last call to **VMSave()**

VMBT_BACKUP a place-holder to keep track of the previous version of a VMBT_DUP block. The uid is the VM block handle to which the file space used to belong.

VMBT_ZOMBIE a block that has been freed since the last **VMSave()**. The handle is preserved in case of a **VMRevert()** (a VMBT_BACKUP block retains the file space).

uid The “used ID” bound to the block.

size Number of bytes allocated for the block in the file.

pos The position of those bytes in the file.

See Also: pgs.

■ pvsiz

Usage: pvsiz [<object>]

Examples:

“pvsiz” print the dimensions of the visual object at *ds:si.

Synopsis: Print out the dimensions of a visual object.

Notes:

- ◆ The object argument is the address to the object to print out. It defaults to *ds:si and is optional. The ‘-i’ flag for an implied grab may be used.
- ◆ The special object flags may be used to specify <object>. For a list of these flags, see pobject.



■ pwd**Usage:** pwd**Examples:** “pwd”**Synopsis:** Prints the current working directory for the current thread.**See Also:** dirs, stdpaths.

■ quit**Usage:** quit [<options>]**Examples:**

“quit cont” continue GEOS and quit swat

“quit det” detach from the PC and quit swat.

Synopsis: Stop the debugger and exit.**Notes:**

- ◆ The <option> argument may be one of the following:
continue: continue GEOS and exit swat;
leave: keep GEOS stopped and exit swat.
- ◆ Anything else causes swat to detach and exit.

See Also: detach.

■ range This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ read-char**Usage:** read-char [<echo>]**Examples:**

“read-char 0”

Read a single character from the user and don't echo it.

Synopsis: Reads a character from the user.**Notes:** If <echo> is non-zero or absent, the character typed will be echoed.**See Also:** read-line.

■ read-line

Usage: read-line [<isTcl> [<initial input> [<special chars>]]]

Examples:

“read-line” reads a single line of text.

“read-line 1” reads a Tcl command.

“read-line 1 {go}”
reads a Tcl command that starts with “go “

“read-line 1 {} {\e\4}”
reads a Tcl command, considering escape and control-d cause
for immediate return, regardless of whether braces and
brackets are balanced

Synopsis: Reads a single line of input from the user. If optional argument is non-zero, the line is interpreted as a Tcl command and will not be returned until all braces/brackets are balanced. The final newline is stripped. Optional second argument is input to be placed in the buffer first. This input must also be on-screen following the prompt, else it will be lost.

Notes:

- ◆ If <isTcl> is non-zero, the input may span multiple lines, as read-line will not return until all braces and brackets are properly balanced, according to the rules of Tcl. This behavior may be overridden by the <special chars> argument.
- ◆ If <initial input> is given and non-empty, it is taken to be the initial contents of the input line and may be edited by the user just as if s/he had typed it in. The string is not automatically displayed; that is up to the caller.
- ◆ <special chars> is an optional string of characters that will cause this routine to return immediately. The character that caused the immediate return is left as the last character of the string returned. You may use standard backslash escapes to specify the characters. This will return even if the user is entering a multi-line Tcl command whose braces and brackets are not yet balanced.
- ◆ The user's input is returned as a single string with the final newline stripped off.



See Also: top-level-read

■ read-reg

Usage: read-reg <register>

Examples:

“read-reg ax”
return the value of AX

“read-reg CC”
return the value of the conditional flags

Synopsis: Return the value of a register in decimal.

Notes:

- ◆ The <register> argument is the two letter name of a register in either upper or lower case.

See Also: frame register, assign, setcc, clrc.

■ regs

Usage: regs

Synopsis: Print the current registers, flags, and instruction.

See Also: assign, setcc, clrc, read-reg.

■ regwin

Usage: regwin [off]

Examples: “regwin”
“regwin off”

Synopsis: Turn the continuous display of registers on or off.

Notes:

- ◆ If you give the optional argument “off”, you will turn off any active register display.
- ◆ If you give no argument, the display will be turned on.
- ◆ Only one register display may be active at a time.

See Also: `display`.

■ `repeatCommand`

Usage: `var repeatCommand <string>`

Examples:

`“var repeatCommand [list foo nil]”`

Execute the command “foo nil” if the user just hits <Enter> at the next command prompt.

Synopsis: This variable holds the command Swat should execute if the user enters an empty command. It is used by all the memory-referencing commands to display the next chunk of memory, and can be used for other purposes as well.

Notes:

- ◆ *repeatCommand* is emptied just before **top-level-read** returns the command the interpreter should execute and must be reset by the repeated command if it wishes to continue to be executed when the user just hits <Enter>.
- ◆ The text of the current command is stored in *lastCommand*, should you wish to use it when setting up *repeatCommand*.

See Also: `target`, `focus`, `mouse`, `keyboard`.

■ `require`

Usage: `require <name> [<file>]`

Examples:

`“require fmtval print”`

Ensure the procedure “fmtval” is defined, loading the file “print.tcl” if it is not.

Synopsis: This ensures that a particular function, not normally invoked by the user but present in some file in the system library, is actually loaded.

Notes: If no <file> is given, a file with the same name (possibly suffixed “.tcl”) as the function is assumed.

See Also: `autoload`.



■ restore-state

Usage: restore-state

Examples:

“restore-state”

Set all registers for the current thread to the values saved by the most recent save-state.

Synopsis: Pops all the registers for a thread from the internal state stack.

Notes:

- ◆ This is the companion to the “save-state” command.
- ◆ All the thread’s registers are affected by this command.

See Also: save-state.

■ ret

Usage: ret [<function name>]

Examples: “ret”
“ret ObjMessage”

Synopsis: Return from a function and stop.

- ◆ The <function name> argument is the name of a function in the patient’s stack after which swat should stop. If none is specified then Swat returns from the current function.
- ◆ The function returned from is the first frame from the top of the stack which calls the function (like the “finish” command).
- ◆ This command does not force a return. The machine continues until it reaches the frame above the function.

See Also: finish, backtrace.

■ **return** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ **return-to-top-level**

Usage: return-to-top-level

Examples:

“return-to-top-level”
Returns to the top-level interpreter.

Synopsis: Forces execution to return to the top-level interpreter loop, unwinding intermediate calls (protected commands still have their protected clauses executed, but nothing else is).

See Also: top-level, protect.

■ **rs**

Usage: rs

Examples:

“rs” restart GEOS without attaching

Synopsis: Restart GEOS without attaching.

See Also: att, attach.

■ **run**

Usage: run [<patient-name>]

Examples:

“run uki”
Run the application with patient name “uki”.

“run -e uki” run EC Uki

“run -n uki” run non-EC Uki

“run -p games\ukiec.geo”
run games\ukiec.geo

“run”
run the default patient, as specified by the patient-default command.



Synopsis: “Runs” an application by loading it via a call to **UserLoadApplication()** and stopping when the app reaches the GenProcess handler for MSG_META_ATTACH. Return patient created, if any (In the examples shown, this would be “uki”).

Notes:

- ◆ May be used even if stopped inside the loader, in which case GEOS will be allowed to continue starting up, and the specified app run after GEOS is Idle.
- ◆ If the machine stops for any other reason other than the call's completion, you are left wherever the machine stopped.

See Also: patient-default, send, spawn, switch.

■ rwatch

Usage: rwatch [(on | off)]

Examples:

“rwatch on” Watch text-recalculation as it happens
 “rwatch off” Turn output off
 “rwatch” See what the status is

Synopsis: Displays information about text recalculation. Specifically designed for tracking bugs in the rippling code.

See Also: ptext.

■ save

Usage: save (<#lines> | <filename>)

Examples:

“save 500” Save the last 500 lines that scroll off the screen.
 “save /dumps/puffball”
 Save the contents of the entire scroll buffer to the file “puffball”.

Synopsis: Controls the scrollbar buffer Swat maintains for its main command window.

Notes:

- ◆ If the argument is numeric, it sets the number of lines to save (the default is 1,000).
- ◆ If the argument is anything else, it's taken to be the name of a file in which the current buffer contents (including the command window) should be saved. If the <filename> is relative, it is taken relative to the directory in which the executable for the patient to which the current stack frame's function belongs is located. If the file already exists, it is overwritten.

■ **save-state**

Usage: save-state

Examples:

“save-state”

Push the current register state onto the thread's state stack.

Synopsis: Records the state of the current thread (all its registers) for later restoration by “restore-state”.

Notes:

- ◆ Swat maintains an internal state stack for each thread it knows, so calling this has no effect on the target PC.
- ◆ This won't save any memory contents, just the state of the thread's registers.

See Also: restore-state, discard-state.

■ **sbwalk**

Usage: sbwalk [<patient>]

Examples:

“sbwalk” list the saved blocks of the current patient.

“sbwalk geos”
list the saved blocks of the GEOS patient.

Synopsis: List all the saved blocks in a patient.



Notes:

- ◆ The <patient> argument is any GEOS patient. If none is specified then the current patient is used.

■ **scan** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ **scope**

Usage: scope [<scope-name>]

Examples:

“scope” Returns the name of the current auxiliary scope.

Synopsis: This changes the auxiliary scope in which Swat looks first when trying to resolve a symbol name in an address expression.

Notes:

- ◆ This command isn’t usually typed by users, but it is the reason you can reference local labels after you’ve listed a function unrelated to the current one.
- ◆ You most likely want to use the set-address Tcl procedure, rather than this command.
- ◆ If <scope-name> is “..”, the auxiliary scope will change to be the lexical parent of the current scope.

See Also: set-address, addr-parse, whatis.

■ **screenwin**

Usage: screenwin

Synopsis: Print the address of the current top-most screen window.

■ send

Usage: send [-enpr] <geode-name>

Examples:

“send icon”

send EC Icon Editor if running in EC mode; send non-EC Icon Editor if running in non-EC mode.

“send -r icon”

send appropriate icon editor, then run it. (See documentation for “run” above.)

“send -e icon”

send EC Icon editor.

“send -n icon”

send non-EC Icon Editor

“send -p c:/pcgeos/appl/icon/icon.geo”

send c:/pcgeos/appl/icon/icon.geo

“send”

send the default patient (as set by the **patient-default** command).

Synopsis: Send a geode from the host to target machine.

■ send-file

Usage: send-file <file> <targfilename>

Examples:

“send-file /pcgeos/appl/sdk/mess1/mess1.geo WORLD/soundapp.geo”

Send the **mess1.geo** file on the host machine to the WORLD directory of the host machine, where it will be called **soundapp.geo**.

Synopsis: Sends a file from the host machine to the target.

■ set-address

Set the last-accessed address recorded for memory-access commands. Single argument is an address expression to be used by the next memory-access command (except via <return>).



■ setcc

Usage: setcc <flag> [<value>]

Examples:

“setcc c” set the carry flag

“setcc z 0” clear the zero flag

Synopsis: Set a flag in the computer.

Notes:

- ◆ The first argument is the first letter of the flag to set. The following is a list of the flags:

t	trap
i	interrupt enable
d	direction
o	overflow
s	sign
z	zero
a	auxiliary carry
p	parity
c	carry

- ◆ The second argument is the value to assign the flag. It defaults to one but may be zero to clear the flag.

See Also: clrc, compcc.

■ set-masks

Usage: set-masks <mask1> <mask2>

Examples:

“set-masks 0xff 0xff”

Allow no hardware interrupts to be handled while the machine is stopped.

Swat Reference

224

Synopsis: Sets the interrupt masks used while the Swat stub is active. Users should use the “int” command.

Notes:

- ◆ <mask1> is the mask for the first interrupt controller, with a 1 bit indicating the interrupt should be held until the stub returns the machine to GEOS. <mask2> is the mask for the second interrupt controller.
- ◆ These masks are active only while the machine is executing in the stub, which usually means only while the machine is stopped.

See Also: int.

■ set-repeat

Sets the command to be repeated using a template string and the lastCommand variable. The variables \$0...\$n substitute the fields 0...n from the lastCommand variable. The final result is placed in repeatCommand which will be executed should the user type Enter.

■ set-startup-ec

Usage: set-startup-ec [<args>]

Examples:

“set-startup-ec +vm”
turn on VM error checking when starting up

“set-startup-ec none”
turn off all ec code when starting up

Synopsis: Executes the “ec” command upon startup, to allow one to override the default error checking flags.

See Also: ec.

■ sftwalk

Usage: sftwalk

Examples: “sftwalk”

Synopsis: Print the SFT out by blocks.



Notes:

- ◆ This is different than sysfiles in that it shows less details of the files and instead shows where the SFT blocks are and what files are in them.

See Also: sysfiles, geosfiles, fwalk.

■ showcalls

Usage: showcalls [<flags>] [<args>]

Examples:

“showcalls -o”
show all calls using ObjMessage and ObjCall*
“showcalls -ml”
show all calls changing global and local memory
“showcalls” stop showing any calls

Synopsis: Display calls to various parts of GEOS.

Notes:

- ◆ The <flags> argument determines the types of calls displayed. Multiple flags must all be specified in the first argument such as “showcalls -vl”. If

no flags are passed then showcalls stops watching. The flags may be any of the following:

- p** Modify all other flags to work for the current patient only
- b** Monitors vis builds
- s** Monitors shutdown: MSG_DETACH, DETACH_COMPLETE, ACK, DETACH_ABORT
- d** Show dispatching of threads
- e** Show FOCUS, TARGET, MODAL, DEFAULT, etc. exclusive grabs & releases
- g** Show geometry manager resizing things (all sizes in hex)
- l** Show local memory create, destroy, relocate
- m** Show global memory alloc, free, realloc
- o** Show **ObjMessage()** and **ObjCall...()**
- w** Show **WinOpen()**, **WinClose()**, **WinMoveResize()**, **WinChangePriority()**.
- N** Show navigation calls (between fields, and between windows)

◆ The<args> argument is used to pass values for some of options.

See Also: mwatch, objwatch.

■ showMethodNames

Usage: var showMethodNames

Synopsis: If this variable is non-zero, Swat prints out the names of the method in the AX register when unassembling a message call.

■ skip

Usage: skip [<number of instructions>]

Examples:

- “skip” skip the current instruction
- “skip 6” skip the next six instructions

Synopsis: Skip one or more instructions.



Notes:

- ◆ The <number of instructions> argument defaults to one if not specified.

See Also: istep, sstep, patch.

■ sleep

Usage: sleep <seconds>

Examples:

“sleep 5” Pauses Swat for 5 seconds.

Synopsis: This pauses Tcl execution for the given number of seconds, or until the user types Ctrl-C.

Notes:

- ◆ Messages from the PC continue to be processed, so a FULLSTOP event will be dispatched if the PC stops, but this command won't return until the given length of time has elapsed.
- ◆ <seconds> is a real number, so “1.5” is a valid argument.
- ◆ Returns non-zero if it slept for the entire time, or 0 if the sleep was interrupted by the user.

■ slist

Usage: slist [<args>]

Examples:

“slist” list the current point of execution

“slist foo.asm::15”
list foo.asm at line 15

“slist foo.asm::15,45”
list foo.asm from lines 15 to 45

Synopsis: List source file lines in swat.

Notes:

- ◆ The args argument can be any of the following:
 - <address> Lists the 10 lines around the given address
 - <line> Lists the given line in the current file
 - <file>:: - <line> Lists the line in the given file
 - <line1>,<line2> Lists the lines between line1 and line2, inclusive, in the current file
 - <file>:: - <line1>,<line2> Lists the range from <file>
- ◆ The default is to list the source lines around CS:IP.

See Also: listi, istep, regs.

■ smatch

Synopsis: Look for symbols of a given class by pattern. First argument <pattern> is the pattern for which to search (it's a standard Swat pattern using shell wildcard characters). Optional second argument <class> is the class of symbol for which to search and is given directly to the "symbol match" command. Defaults to "any".

■ source This is a Tcl primitive. See "Tool Command Language," Chapter 5.

■ sort

Usage: sort [-r] [-n] [-u] <list>

Examples:

 "sort -n \$ids"
 Sorts the list in \$ids into ascending numeric order.

Synopsis: This sorts a list into ascending or descending order, lexicographically or numerically.



Notes:

- ◆ If “-r” is given, the sort will be in descending order.
- ◆ If “-u” is given, duplicate elements will be eliminated.
- ◆ If “-n” is given, the elements are taken to be numbers (with the usual radix specifiers possible) and are sorted accordingly.
- ◆ The sorted list is returned.

See Also: map, foreach, mapconcat.

■ spawn

Usage: spawn <processName> [<addr>]

Synopsis: Set a temporary breakpoint in a not-yet-existent process/thread, waiting for a new one to be created. First argument is the permanent name of the process to watch for. Second argument is an address expression specifying where to place the breakpoint. If no second argument is present, the machine will be stopped and Swat will return to the command level when the new thread is spawned by GEOS.

Notes:

- ◆ This can also be used to catch the spawning of a new thread.
- ◆ If the machine stops before the breakpoint can be set, you’ll have to do this again.

■ src

This is a Tcl primitive. See page 316.

■ srcwin

Usage: srcwin <numLines>

Examples:

“srcwin 6” Show 6 lines of source context around CS:IP
 “srcwin 0” Show no source lines, i.e. turn the display off.

Synopsis: Set the number of lines of source code to be displayed when the target machine stops.

Notes:

- ◆ Only one source display may be active at a time.

See Also:

display, regwin, search.

■ **sstep**

Usage: sstep [<default command>]

Examples:

“ss” enter source step mode

“sstep n” enter source step mode, <ret> does a next command

Synopsis:

Step through the execution of the current patient by source lines. This is THE command for stepping through high-level (e.g., C) code.

- ◆ The <default> command argument determines what pressing the <Return> key does. By default, <Return> executes a step command. Any other command listed below may be substituted by passing the letter of the command.
- ◆ Sstep steps through the patient line by line, printing where the instruction pointer is and what line is to be executed Sstep waits for the user to type a command which it performs and then prints out again where sstep is executing.
- ◆ This is a list of sstep commands:

q, <Esc>, ‘ ‘ Stops sstep and returns to command level.

b Toggles a breakpoint at the current location.

c Stops sstep and continues execution.

n Continues to the next source line, skipping procedure calls, repeated string instructions, and software interrupts. Only stops when the machine returns to the right context (i.e. the



stack pointer and current thread are the same as they are when the 'n' command was given).

- I** Goes to the next library routine.
- N** Like n, but stops whenever the breakpoint is hit, whether you're in the same frame or not.
- M** Goes to the next message called. Doesn't work when the message is not handled anywhere.
- f** Finishes out the current stack frame.
- s, <Ret>** Steps one source line
- S** Skips the current instruction
- J** Jump on a conditional jump, even when "Will not jump" appears. This does not change the condition codes.
- g** Executes the 'go' command with the rest of the line as arguments.
- e** Executes a Tcl command and returns to the prompt.
- R** References either the function to be called or the function currently executing.
- h, ?** A help message.

- ◆ Emacs will load in the correct file executing and following the lines where sstep is executing if its server is started and if ewatch is on in swat. If ewatch is off emacs will not be updated.
- ◆ If the current patient isn't the actual current thread, sstep waits for the patient to wake up before single-stepping it.

See Also: istep, listi, ewatch.

■ stdpaths

Usage: stdpaths

Examples: "stdpaths"

Synopsis: Print out all paths set for standard directories

See Also: pwd, dirs.



■ step

Usage: step

Examples:

“step” execute the next instruction

“s”

Synopsis: Execute the patient by a single machine instruction.

Notes:

- ◆ If waitForPatient is non-zero, step waits for the machine to stop again.
- ◆ This doesn't do any of the checks for special conditions (XchgTopStack, software interrupts, etc.) performed by the 's' command in istep.

See Also: istep, next.

■ step-patient

Usage: step-patient

Examples:

“step-patient”
Execute a single instruction on the target PC.

Synopsis: Causes the PC to execute a single instruction, returning only when the instruction has been executed.

Notes:

- ◆ Unlike the continue-patient command, this command will not return until the machine has stopped again.
- ◆ No other thread will be allowed to run, as timer interrupts will be turned off while the instruction is being executed.

See Also: help-fetch.



■ step-until

Usage: step-until expression [byte | word]

Examples:

“step-until ax=0”

Single-step until ax is zero.

“step-until ds:20h!=0 byte”

Single-step until byte at ds:20h is non-zero

“step-until ds:20h!=0 word”

Single-step until word at ds:20h is non-zero

“step-until c=0”

Single-step until the carry is clear

“step-until ax!=ax”

Step forever

This command causes Swat to step until a condition is met.

Notes: Useful for tracking memory or register trashing bugs.

See Also: step-while

■ stop

Usage: stop in <class>::<message> [if <expr>]
 stop in <procedure> [if <expr>]
 stop in <address-history-token> [if <expr>]
 stop at [<file>:]<line> [if <expr>]
 stop <address> [if <expr>]

Examples: “stop in main”
 “stop in @3”
 “stop at /staff/pcgeos/Loader/main.asm:36 if { joe_local ==22}”
 “stop at 25”
 “stop MemAlloc+3 if {ax==3}”

Synopsis: Specify a place and condition at which the machine should stop executing. This command is intended primarily for setting breakpoints when debugging a geode created in C or another high-level language, but may also be used when debugging assembly-language geodes.

Notes:

- ◆ “stop in” will set a breakpoint at the beginning of a procedure, immediately after the procedure’s stack frame has been set up.
- ◆ “stop at” will set a breakpoint at the first instruction of the given source line. If no <file> is specified, the source file for the current stack frame is used.
- ◆ If a condition is specified, by means of an “if <expr>” clause, you should enclose the expression in {}’s to prevent any nested commands, such as a “value fetch” command, from being evaluated until the break-point is hit.
- ◆ For convenience, “stop in” also allows address-history tokens. This is useful when used in conjunction with the “methods” command.

See Also: brk, ibrk

■ stop-catch

Usage: stop-catch <body>

Examples:

“stop-catch {go ProcCallModuleRoutine}”

Let machine run until it reaches **ProcCallModuleRoutine()**, but do not issue a FULLSTOP event when it gets there.

Synopsis: Allows a string of commands to execute without a FULLSTOP event being generated while they execute.

Notes: Why is this useful? A number of things happen when a FULLSTOP event is dispatched, including notifying the user where the machine stopped. This is inappropriate in something like “istep” or “cycles” that is single-stepping the machine, for example.

See Also: event, continue-patient, step-patient.

■ stop-patient

Usage: stop-patient

Examples:

“stop-patient”

Stops the target PC.



- Synopsis:** Stops the target PC, in case you continued it and didn't wait for it to stop on its own.
- Notes:** This is different from the "stop" subcommand of the "patient" command.
- See Also:** continue-patient.

■ stream

Usage:

```
stream open <file> (r | w | a | r+ | w+)
stream read (line | list | char) <stream>
stream print <list> <stream>
stream write <string> <stream>
stream rewind <stream>
stream seek (<posn> | +<incr> | -<decr> | end) <stream>
stream state <stream>
stream eof <stream>
stream close <stream>
stream flush <stream>
stream watch <stream> <what> <procName>
stream ignore <stream>
```

Examples:

```
"var s [stream open kmap.def w]"
    Open the file "kmap.def" for writing, creating it if it wasn't
    there before, and truncating any existing file.
```

```
"stream write $line $s"
    Write the string in $line to the open stream.
```

Synopsis: This allows you to read, write, create, and otherwise manipulate files on the host machine from Swat.

Notes:

- ◆ Subcommands may be abbreviated uniquely.
- ◆ Streams are a precious resource, so you should be sure to always close them when you are done. This means stream access should usually be performed under the wings of a "protect" command so the stream gets closed even if the user types Ctrl+C.
- ◆ Swat's current directory changes as you change stack frames, with the directory always being the one that holds the executable file for the

patient to which the function in the current frame belongs. If the <file> given to “stream open” isn’t absolute, it will be affected by this.

- ◆ The global variable `file-init-dir` contains the absolute path of the directory in which Swat was started. It can be quite useful when forming the <file> argument to “stream open”.
- ◆ The second argument to “stream open” is the access mode of the file. The meanings of the 5 possible values are:
 - r** read-only access. The <file> must already exist.
 - w** write-only access. If <file> doesn’t already exist, it will be created. If it does exist, it will be truncated.
 - a** append mode. The file is opened for writing only. If <file> doesn’t already exist, it will be created. If it does exist, writing will commence at its end.
 - r+** read/write. The <file> must already exist. A single read/write position is maintained, and it starts out at the start of the file.
 - w+** read/write. If <file> doesn’t already exist, it will be created. If it does exist, it will be truncated. A single read/write position is maintained, and it starts out at the start of the file.
- ◆ “stream read” can read data from the stream in one of three formats:
 - line** Returns all the characters from the current position up to the first newline or the end of the file, whichever comes first. The newline, if seen, is placed at the end of the string as `\n`. Any other non-printable characters or backslashes are similarly escaped.
 - list** Reads a single list from the stream, following all the usual rules of Tcl list construction. If the character at the current read position is a left brace, this will read to the matching right brace, bringing in newlines and other whitespace. If there is whitespace at the initial read position, it is skipped. Standard Tcl comments before the start of the list are also skipped over (so if the first non-whitespace character encountered is `#`, the



characters up to the following newline or end-of-file will also be skipped).

char This reads a single character from the stream. If the character isn't printable ASCII, it will be returned as one of the regular Tcl backslash escapes.

If there's nothing left to read, you will get an empty string back.

- ◆ “stream write” writes the string exactly as given, without interpreting backslash escapes. If you want to include a newline or something of the sort in the string, you'll need to use the “format” command to generate the string, or place the whole thing in braces and have the newlines in there literally.
- ◆ While the syntax for “stream print” is the same as for “stream write”, there is a subtle difference between the two. “stream write” will write the string as it's given, while “stream print” is intended to write out data to be read back in by “stream read list”. Thus the command

```
stream write {foo biff} $s
```

would write the string “foo biff” to the stream. In contrast,

```
stream print {foo biff} $s
```

would write “{foo biff}” followed by a newline.

- ◆ To ensure that all data you have written has made it to disk, use the “stream flush” command. Nothing is returned.
- ◆ “stream rewind” repositions the read/write position at the start of the stream. “stream seek” gives you finer control over the position. You can set the stream to an absolute position (obtained from a previous call to “stream seek”) by passing the byte number as a decimal number. You can also move forward or backward in the file a relative amount by specifying the number of bytes to move, preceded by a “+”, for forward, or a “-”, for backward. Finally, you can position the pointer at the end of the file by specifying a position of “end”.
- ◆ “stream seek” returns the new read/write position, so a call of “stream seek +0 \$s” will get you the current position without changing anything. If the seek couldn't be performed, -1 is returned.
- ◆ “stream state” returns one of three strings: “error”, if there's been some error accessing the file, “eof” if the read/write position is at the end of the

file, or “ok” if everything’s fine. “stream eof” is a shortcut for figuring if you’ve reached the end of the file.

- ◆ “stream close” shuts down the stream. The stream token should never be used again.
- ◆ “stream watch” and “stream ignore” are valid only on UNIX and only make sense if the stream is open to a device or a socket. “stream watch” causes the procedure <procName> to be called whenever the stream is ready for the access indicated by <what>, which is a list of conditions chosen from the following set:

read the stream has data that may be read.

write the stream has room for data to be written to it.

When the stream is ready, the procedure is called:

<procName> <stream> <what>

where <what> is the list of operations for which the stream is ready.

See Also: protect, source, file.



■ **string** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ **switch**

Usage: `switch <thread-id>`
 `switch [<patient>] [:<thread-num>]`

Examples:

“switch 3730h”

Switches swat’s current thread to be the one whose handle ID is 3730h.

“switch :1” Switches Swat’s current thread to be thread number 1 for the current patient.

“switch parallel:2”

Switches Swat’s current thread to be thread number 2 for the patient “parallel”

“switch write”

Switches Swat’s current thread to be thread number 0 (the process thread) for the patient “write”

“switch” Switches Swat’s current thread to be the current thread on the PC.

Synopsis: Switches between applications/threads.

Notes:

- ◆ Takes a single argument of the form <patient>:<thread-num> or <threadID>. With the first form, :<thread-num> is optional -- if the patient has threads, the first thread is selected. To switch to another thread of the same patient, give just :<thread-num>. You can also switch to a patient/thread by specifying the thread handle ID. NOTE: The switch doesn’t happen on the PC—just inside swat.
- ◆ If you don’t give an argument, it switches to the actual current thread in the PC.

■ **symbol** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ **symbolCompletion**

Usage: var symbolCompletion [(0 | 1)]

Examples:

“var symbolCompletion 1”
Enable symbol completion in the top-level command reader.

Synopsis: This variable controls whether you can ask Swat to complete a symbol for you while you’re typing a command. Completion is currently very slow and resource-intensive, so you probably don’t want to enable it.

Notes:

- ◆ Even when symbolCompletion is 0, file-name, variable-name, and command- name completion are always enabled, using the keys described below.
- ◆ When completion is enabled, three keys cause the interpreter to take the text immediately before the cursor and look for all symbols that begin with those characters. The keys are:

Ctrl+D Produces a list of all possible matches to the prefix.

Escape Completes the command as best possible. If the characters typed so far could be the prefix for more than one command, Swat will fill in as many characters as possible.

Ctrl+] Cycles through the list of possible symbols, in alphabetical order.

See Also: top-level-read.

■ **sym-default**

Usage: sym-default [<name>]

Examples:

“sym-default motif”
Make swat look for any unknown symbols in the patient named “motif” once all other usual places have been searched.



Synopsis: Specifies an additional place to search for symbols when all the usual places have been searched to no avail.

Notes:

- ◆ The named patient need not have been loaded yet when you execute this command.
- ◆ A typical use of this is to make whatever program you're working on be the sym-default in your .swat file so you don't need to worry about whether it's the current one, or reachable from the current one, when the machine stops and you want to examine the patient's state.
- ◆ If you don't give a name, you'll be returned the name of the current sym-default.

■ sysfiles

Usage: sysfiles

Examples: "sysfiles"

Synopsis: Print out all open files from dos's system file table.

Notes: Normally SFT entries that aren't in-use aren't printed. If you give the optional argument "all", however, all SFT entries, including those that aren't in-use, will be printed.

See Also: geosfiles, sftwalk, fwalk.

■ systemobj

Usage: systemobj

Examples:

"gentree [systemobj]"
print the generic tree starting at the system's root

"pobject [systemobj]"
print the system object

Synopsis: Prints out the address of the uiSystemObj, which is the top level of the generic tree.

Notes:

- ◆ This command is normally used with `gentree` as shown above to print out the whole generic tree starting from the top.

See Also: `gentree`, `impliedgrab`.

■ table	This is a Tcl primitive. See page 321.
----------------	--

target

Usage: `target [<object>]`

Examples:

`"target"` print target hierarchy from the system object down
`"target -i"` print target hierarchy from implied grab down
`"target ^l4e10h:20h"`
 print target hierarchy from ^l4e10h:20h down
`"target [content]"`
 print target hierarchy from content under mouse.

Synopsis: Prints the target hierarchy below an object.

Notes:

- ◆ If no argument is specified, the system object is used.
- ◆ The special object flags may be used to specify `<object>`. For a list of these flags, see `pobject`.
- ◆ Remember that the object you start from may have the target within its part of the hierarchy, but still not have the target because something in a different part of the tree has it.
- ◆ The variable `"printNamesInObjTrees"` can be used to print out the actual app-defined labels for the objects, instead of the class, where available.
This variable defaults to false.

See Also: `focus`, `model`, `mouse`, `keyboard`, `pobject`.



■ targetobj

Usage: targetobj

Examples:

“targetobj” return object with target
 “pobj [targetobj]”
 do a pobject on the target object (equivalent to “pobj -t”).

Synopsis: Returns the object with the target.

See Also: target, focus, focusobj, modelobj.

■ tbrk

Usage: tbrk <addr> <condition>*
 tbrk del <tbrk>+
 tbrk list
 tbrk cond <tbrk> <condition>*
 tbrk count <tbrk>
 tbrk reset <tbrk>
 tbrk address <tbrk>

Examples:

“tbrk ObjCallMethodTable”
 Count the number of times ObjCallMethodTable() is called.

“tbrk count 2”
 Find the number of times tally breakpoint number 2 was hit.

“tbrk reset 2”
 Reset the counter for tbrk number 2 to 0.

“tbrk list” Print a list of the set tally breakpoints and their current counts.

Synopsis: This command manipulates breakpoints that tally the number of times they are hit without stopping execution of the machine—the breakpoint is noted and the machine is immediately continued. Such a breakpoint allows for real-time performance analysis, which is nice.

Notes:

- ◆ If you specify one or more <condition> arguments when setting the tally breakpoint, only those stops that meet the conditions will be counted.
- ◆ The *condition* argument is exactly as defined by the “brk” command, q.v..
- ◆ When you’ve set a tally breakpoint, you will be returned a token of the form “tbrk<n>”, where <n> is some number. You use this token, or just the <n>, if you’re not a program, wherever <tbrk> appears in the Usage description, above.
- ◆ There are a limited number of tally breakpoints supported by the stub. You’ll know when you’ve set too many.
- ◆ “tbrk address” returns the address at which the tbrk was set, as a symbolic address expression.

See Also: brk, cbrk.

■ tcl-debug

Usage:

```
tcl-debug top
tcl-debug next <tcl-frame>
tcl-debug prev <tcl-frame>
tcl-debug args <tcl-frame>
tcl-debug getf <tcl-frame>
tcl-debug setf <tcl-frame> <flags>
tcl-debug eval <tcl-frame> <expr>
tcl-debug complete <tcl-frame>
tcl-debug next-call
```

Examples:

```
“var f [tcl-debug top]”
    Sets $f to be the frame at which the debugger was entered.

“var f [tcl-debug next $f]”
    Retrieves the next frame down (away from the top) the Tcl call
    stack from $f.
```

Synopsis: This provides access to the internals of the Tcl interpreter for the Tcl debugger (which is written in Tcl, not C). It will not function except after the debugger has been entered.

See Also: debug.



■ text-fixup

Usage:

1 Run geos under swat, run swat on the development system

2 Run GeoWrite

3 Open the GeoWrite file that needs fixing

4 Set the breakpoint in swat:

```
patch text::CalculateRegions
=> text-fixup
```

This will set a breakpoint at the right spot

5 Turn on the error-checking code in swat:

```
ec +text
```

6 Enter a <space> into the document. This forces recalculation which will cause **CalculateRegions()** to be called which will cause text-fixup to be called.

If it worked, this code should patch together the file. If it's not, you'll get a FatalError right now.

7 Turn off the ec code and disable the fixup breakpoint.

```
ec none
dis <breakpoint number>
continue
```

8 Delete the space and save the file.

To do another file, you can just enable the breakpoint once the new file is open and turn on the ec code.

Synopsis: Helps fix up trashed GeoWrite documents.

■ thaw

Usage:

```
thaw [<patient>]
thaw :<n>
```



See Also: ps.

■ timebrk

Usage: timebrk <start-addr> <end-addr>+
timebrk del <timebrk>+
timebrk list
timebrk time <timebrk>
timebrk reset <timebrk>

Examples:

“timebrk LoadResourceData -f”

Calculate the time required to process a call to **LoadResourceData()**.

“timebrk time 2”

Find the amount of time accumulated for timing breakpoint number 2.

“timebrk reset 2”

Reset the counter for timebrk number two to zero.

“timebrk list”

Print a list of the set timing breakpoints and their current counts and time.

Synopsis: This command manipulates breakpoints that calculate the amount of time executing between their starting point and a specified ending point. The breakpoints also record the number of times their start is hit, so you can figure the average amount of time per hit.

Notes:

- ◆ You can specify a place at which timing should end either as an address or as “-f”. If you use “-f”, timing will continue until the finish of the routine at whose start you’ve placed the breakpoint. Such a breakpoint may only be set at the start of a routine, as the stub hasn’t the

wherewithal to determine what the return address is at an arbitrary point within the function.

- ◆ You may specify more than one ending point. Timing will stop when execution reaches any of those points.
- ◆ When you've set a timing breakpoint, you will be returned a token of the form "timebrk<n>", where <n> is some number. You use this token, or just the <n>, if you're not a program, wherever <timebrk> appears in the Usage description, above.

See Also: brk, cbrk, tbrk.

■ timingProcessor

Usage: var timingProcessor [i86 | i88 | i286 | V20]

Synopsis: The processor for which to generate cycle counts.

■ tmem

Usage: tmem

Examples:

"tmem" turn on memory tracing.

Synopsis: Trace memory usage.

Notes: The tmem command catches calls to DebugMemory, printing out the parameters passed (move, free, realloc, discard, swapout, swapin, modify).

■ top-level

Usage: top-level

Examples:

"top-level" Begin reading and interpreting Tcl commands in a nested interpreter.

Synopsis: This is the top-most read-eval-print loop of the Swat Tcl interpreter.

Notes: This command will only return if the user issues the "break" command. Otherwise it loops infinitely, reading and executing and printing the results of Tcl commands.

See Also: top-level-read.



■ tundocalls

Usage: tundocalls [-acPCrR]

Examples:

“tundocalls -a” Print out all text undo calls

“tundocalls -r” Print run undo calls

“tundocalls -R” Print replace undo calls

“tundocalls -c” Print info when undo information is created

“tundocalls -cP” Print info about para attributes only

“tundocalls -cC” Print info about char attributes only

“tundocalls”

Synopsis: Prints out information about each undo call made to the text object.

See Also: ptext, showcalls.

■ twalk

Usage: twalk

Examples:

“twalk” print all the timers in the system.

“twalk -o ui” print all the timers in the system for the ui thread.

“twalk -a” print all the timers with the “real” data for the time for time remaining rather than maintaining a total.

Synopsis: List all the timers in GEOS.

■ type This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ unalias

Usage: unalias <name>+

Examples:

“unalias p” Removes “p” as an alias for print.

Synopsis: This removes an alias.

Notes:

- ◆ In fact, this actually can be used to delete any command at all, including Tcl procedures and Swat built-in commands. Once they're gone, however, there's no way to get them back.

See Also: alias.

■ unassemble

Usage: unassemble [<addr> [<decode-args>]]

Examples:

“unassemble cs:ip 1”

Disassemble the instruction at CS:IP and return a string that shows the values of the arguments involved.

Synopsis: This decodes data as machine instructions and returns them to you for you to display as you like. It is not usually typed from the command line.

Notes:

- ◆ The return value is always a four-element list:

```
{<symbolic-addr> <instruction> <size> <args>}
```

where <symbolic-addr> is the address expressed as an offset from some named symbol, <instruction> is the decoded instruction (without any leading whitespace), <size> is the size of the instruction (in bytes) and <args> is a string displaying the values of the instruction operands, if <decode-args> was given and non-zero (it is the empty string if <decode-args> is missing or 0).
- ◆ If <addr> is missing or “nil”, the instruction at the current frame's CS:IP is returned.

See Also: listi.

■ unbind-key

Usage: unbind-key <ascii_value>

Examples:

“unbind-key \321”

Unbinds scroll-down key on host machine.



Synopsis: Unbinds the passed ASCII value.

See Also: alias, bind-key, get-key-binding.

■ **undebug**

Usage: undebug <proc-name>+

Examples:

“undebug fooproc”
Cease halting execution each time “fooproc” is executing.

Synopsis: Removes a Tcl breakpoint set by a previous “debug” command.

See Also: debug.

■ **up**

Usage: up [<frame offset>]

Examples:

“up” move the frame one frame up the stack
“up 4” move the frame four frames up the stack

Synopsis: Move the frame up the stack.

Notes:

- ◆ The <frame offset> argument is the number of frame to move up the stack. If none is specified then the current frame is moved up one frame.
- ◆ This command may be repeated by pressing <Return>.

See Also: backtrace, down.

■ **value**

Usage: value fetch <addr> [<type>]
value store <addr> <value> [<type>]
value hfetch <num>

```
value hstore <addr-list>
value hset <number-saved>
```

Examples:

```
“value fetch ds:si [type word]”
    Fetch a word from ds:si

“value store ds:si 0 [type word]”
    Store 0 to the word at ds:si

“value hfetch 36”
    Fetch the 36th address list stored in the value history.

“value hstore $a”
    Store the address list in $a into the value history.

“value hset 50”
    Keep track of up to 50 address lists in the value history.
```

Synopsis:

This command allows you to fetch and alter values in the target PC. It is also the maintainer of the value history, which you normally access via @<number> terms in address expressions.

Notes:

- ◆ “value fetch” returns a value list that contains the data at the given address. If the address has an implied data type (it involves a named variable or a structure field), then you do not need to give the <type> argument.

All integers and enumerated types are returned in decimal. 32-bit pointers are returned as a single decimal integer whose high 16 bits are the high 16 bits (segment or handle) of the pointer. 16-bit pointers are likewise returned as a single decimal integer.

Characters are returned as characters, with non-printable characters converted to the appropriate backslash escapes (for example, newline is returned as \n).

Arrays are returned as a list of value lists, one element per element of the array.

Structures, unions and records are returned as a list of elements, each of which is a 3-element list: {<field-name> <type> <value>} <field-name> is the name of the field, <type> is the type token for the type of data stored



in the field, and <value> is the value list for the data in the field, appropriate to its data type.

- ◆ You will note that the description of value lists is recursive. For example, if a structure has a field that is an array, the <value> element in the list that describes that particular field will be itself a list whose elements are the elements of the array. If that array were an array of structures, each element of that list would again be a list of {<field-name> <type> <value>} lists.
- ◆ The “field” command is very useful when you want to extract the value for a structure field from a value list.
- ◆ As for “value fetch”, you do not need to give the <type> argument to
- ◆ “value store” if the <addr> has an implied data type. The <value> argument is a value list appropriate to the type of data being stored, as described above.
- ◆ “value hstore” returns the number assigned to the stored address list. These numbers always increase, starting from 1.
- ◆ If no address list is stored for a given number, “value hfetch” will generate an error.
- ◆ “value hset” controls the maximum number of address lists the value history will hold. The value history is a FIFO queue; if it holds 50 entries, and the 51st entry is added to it, the 1st entry will be thrown out.

See Also: addr-parse, assign, field.

■ **var** This is a Tcl primitive. See “Tool Command Language,” Chapter 5.

■ **varwin**

Usage: varwin <num-lines> <var-name>

■ **view**

Usage: view [<args>]

Examples:

“view foo.goc”

Bring up **foo.goc** in the source window.

Synopsis: View a file in Swat.



See Also: view-line, view-default, srcwin.

■ view-default

Usage: view-default [patient]

Examples:

“view-default spool”
sets the default view to the spool patient.

“view-default”
turns off the view default.

Synopsis: If the view-default is set the view command will automatically look for source files from that patient. If it's not set then the view command will look for files from the current patient.

See Also: view, view-line, srcwin.

■ view-size

Usage: view-size <number-of-lines>

Examples:

“view-size 10”
Makes the view window 10 lines high.

See Also: view, view-line, view-default, srcwin.

■ vistree

Usage: vistree [<address>] [<instance field>]

Examples:

“vistree” print the visual tree starting at *DS:SI

“vistree -i” print the visual tree under the mouse

“vistree @23 VI_optFlags”
print the visual tree with opt flags

“vistree *uiSystemObj”
starts the visual tree at the root of the system.

Synopsis: Print out a visual tree.



Notes:

- ◆ The <address> argument is the address to an object in the generic tree. This defaults to *DS:SI. The '-i' flag for an implied grab may be used.
- ◆ The special object flags may be used to specify <object>. For a list of these flags, see pobject.
- ◆ The <instance field> argument is the offset to any instance data within the VisInstance which should be printed out.
- ◆ The variable "printNamesInObjTrees" can be used to print out the actual app-defined labels for the objects, instead of the class, where available. This variable defaults to false.

See Also: vup, gentree, impliedgrab, pobject.

■ vup

Usage: vup [<address>] [<instance field>]

Examples:

```
"vup"           print the visual object at *DS:SI and its ancestors
"vup @23 VI_optFlags"
                  print the states of object @23 and its ancestors
"vup -i"         print the visual object under the mouse and the object's
                  ancestors
```

Synopsis: Print a list of the object and all of its visual ancestors.

Notes:

- ◆ The <address> argument is the address to an object in the visual tree. This defaults to *DS:SI. The '-i' flag for an implied grab may be used.
- ◆ The special object flags may be used to specify <object>. For a list of these flags, see pobject.
- ◆ The <instance field> argument is the offset to any instance data within the GenInstance which should be printed out.

See Also: vistree, gup, gentree, impliedgrab, pobject.

■ wait

Usage: wait

Examples:

“wait” Wait for the target PC to halt.

Synopsis: This is used after the machine has been continued with “continue-patient” to wait for the machine to stop again. Its use is usually hidden by calling “cont” or “next”.

Notes:

- ◆ This returns 0 if the patient halted naturally (because it hit a breakpoint), and 1 if it was interrupted (by the user typing Ctrl+C to Swat).
- ◆ Most procedures won’t need to use this function.

See Also: brk, ibrk

■ waitForPatient

Usage: var waitForPatient [(1 | 0)]

Examples:

“var waitForPatient 0”
Tells Swat to return to the command prompt after continuing the machine.

Synopsis: Determines whether the command-level patient-continuation commands (step, next, and cont, for example) will wait for the machine to stop before returning.

Notes:

- ◆ The effect of this is to return to the command prompt immediately after having issued the command. This allows you to periodically examine the state of the machine without actually halting it.
- ◆ The output when the machine does stop (e.g. when it hits a breakpoint) can be somewhat confusing. Furthermore, this isn’t fully tested, so it should probably be set to 0 only in somewhat odd circumstances.

See Also: step, next, cont, int.



■ wakeup

Wait for a given patient/thread to wake up. WHO is of the same form as the argument to the “switch” command, (“help switch” to find out more). Leaves you stopped in the kernel in the desired thread’s context unless something else causes the machine to stop before the patient/thread wakes up. WHO defaults to the current thread.

■ wakeup-thread

Subroutine to actually wake up a thread. Argument WHO is as for the “switch” command. Returns non-zero if the wakeup was successful and zero if the machine stopped for some other reason.

■ wclear

Usage: wclear

Synopsis: Clears the current window.

■ wcreate

Usage: wcreate <height>

Synopsis: Create a window of the given height and return a token for it. The window is placed just above the command window, if there’s room. If there aren’t that many lines free on the screen, an error is returned.

■ wdelete

Usage: wdelete <window>

Synopsis: Delete the given window. All windows below it move up and the command window enlarges.

■ whatat

Usage: whatat [<address>]

Examples:

“whatat” name of variable at *DS:SI

“whatat ^12ef0h:002ah”
 name of variable at the specified address

Synopsis: Print the name of the variable at the address.

Notes:

- ◆ The <address> argument specifies where to find a variable name for. The address defaults to *DS:SI.
- ◆ If no appropriate variable is found for the address, ‘*nil*’ is returned.

See Also: pobject, hwalk, lhwalk.

■ whatis

Usage: whatis (<symbol> | <addr>)

Examples: “whatis WinColorFlags”

Synopsis: This produces a human-readable description of a symbol, giving whatever information is pertinent to its type.

Notes:

- ◆ For type symbols (e.g. structures and enumerated types), the description of the type is fully displayed, so if a structure has a field with an enumerated type, all the members of the enumerated type will be printed as well. Also all fields of nested structures will be printed. If this level of detail isn’t what you need, use the “pscope” command instead.
- ◆ It’s not clear why you’d need the ability to find the type of an address-expression, since those types always come from some symbol or other, but if you want to type more, you certainly may.

■ where

Common alias for “backtrace”

■ why

Usage: why

Examples: “why”

Synopsis: Print a description of why the system crashed.



Notes:

- ◆ This must be run from within the frame of the FatalError function. Sometimes GEOS is not quite there. In this case, step an instruction or two and then try the “why” command again.
- ◆ This simply looks up the enumerated constant for the error code in AX in the “FatalErrors” enumerated type defined by the geode that called FatalError. For example, if a function in the kernel called FatalError, AX would be looked up in `geos::FatalErrors`, while if a function in your application called FatalError, this function would look it up in the FatalErrors type defined by your application. Each application defines this enumerated type by virtue of having included **ec.def** or **ec.goh**.
- ◆ For certain fatal errors, additional information is provided by invoking the command `<patient>::<error code>`, if it exists.

See Also: regs, backtrace, explain.

■ wintree

Usage: wintree <window handle> [<data field>]

Examples:

“wintree ^hd060h”
 print a window tree starting at the handle d060h

Synopsis: Print a window tree starting with the root specified.

Notes:

- ◆ The <window address> argument is the address to a window.
- ◆ The <data field> argument is the offset to any instance data within a window (like `W_ptrFlags`).

See Also: vistree, gentree.

■ winverse

Usage: winverse

Synopsis: Sets the inverse-mode of the current window (whether newly-echoed characters are displayed in inverse video) on or off, depending on its argument (1 is on).

■ wmove

Usage: wmove [(+|-)] <x-coord> [(+|-)] <y-coord>

Synopsis: Moves the cursor for the current window. Takes two arguments: the new *x* position and the new *y* position. These positions may be absolute or relative (absolute positions begin with + or -). If you attempt to move outside the current window, an error is generated. This command returns the new cursor position as {*x y*}.

■ words

Usage: words [<address>] [<length>]

Examples:

“words” lists 8 words at DS:SI

“words ds:di 16”
lists 16 words starting at DS:DI

Synopsis: Examine memory as a dump of words.

Notes:

- ◆ The <address> argument is the address to examine. If not specified, the address after the last examined memory location is used. If no address has been examined then DS:SI is used for the address.
- ◆ The <length> argument is the number of bytes to examine. It defaults to 8.
- ◆ Pressing <Return> after this command continues the list.

See Also: bytes, dwords, imem, assign.

■ wpop

Usage: wpop

Synopsis: Revert the current window to its previously pushed value.

■ wpush

Usage: wpush <window>



Synopsis: Switch to a new window, saving the old current-window. Use **wpop** to go back to the previous window. All I/O goes through the current window.

■ **wrefresh**

Usage: wrefresh

Synopsis: Synchronizes the current window with the screen. This need only be performed if you don't echo a newline, as echoing a newline refreshes the current window.

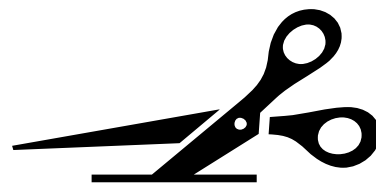
■ **wtop**

Usage: wtop <flag>

Synopsis: Sets where windows go. If argument is non-zero, windows go at the top of the screen and work down. Else windows go at the bottom of the screen and work up



Tool Command Language



5

5.1	Using This Chapter	265
5.2	Copyright Information	266
5.3	Background and Description.....	266
5.4	Syntax and Structure.....	267
5.4.1	Basic Command Syntax.....	268
5.4.1.1	Comments.....	268
5.4.1.2	Argument Grouping.....	268
5.4.1.3	Command Grouping.....	269
5.4.1.4	Command Substitution.....	270
5.4.1.5	Variable Substitution.....	270
5.4.1.6	Backslash Substitution.....	271
5.4.2	Expressions.....	273
5.4.3	Lists.....	275
5.4.4	Command Results	275
5.4.5	Procedures	276
5.4.6	Variables.....	277
5.5	Commands	277
5.5.1	Notation	277
5.5.2	Built-in Commands	278
5.6	Coding	302
5.6.1	Swat Data Structure Commands	303
5.6.2	Examples	328
5.7	Using a New Command.....	330
5.7.1	Compilation	331
5.7.2	Autoloading	331
5.7.3	Explicit Loading.....	331





Before using this chapter one should have a good understanding of how the Swat commands function “Swat Introduction,” Chapter 3, and of how the GEOS system works as a whole.

This chapter is designed to provide information about the Tool Command Language, abbreviated Tcl, (the language in which Swat commands are written) so that new commands can be written and old commands modified. This chapter contains the following main sections: 5.1

- ◆ **Using This Chapter**
Discussion of the situations warranting the construction of a new command in the Tool Command Language.
- ◆ **Background and Description**
Discussion of history of Tool Command Language and general overview of the language.
- ◆ **Syntax and Structure**
Description of the syntax and structure of the Tool Command Language.
- ◆ **Commands**
List of all built-in commands for the language.
- ◆ **Coding**
Descriptions and examples of coding conventions, techniques, and tricks.
- ◆ **Installation**
Steps to take in order to be able to use a newly written command to help debug an application.

5.1 Using This Chapter

This chapter provides the information needed to write a new Swat command in Tcl. But, new commands need only be written in certain situations. Some of the situations in which it is advantageous to write a new Swat command in Tcl are:

- ◆ There is complex task that is being repeated often. For example, if one is continually examining a certain piece of data in memory but has to go



through many steps to do so, then it is helpful to write a single command to perform all of the needed steps.

- ◆ When a new data-structure is created for an application. For example, if one creates a look-up table for the application, then a Tcl command should be written to examine that table in particular.

The existing Swat commands should take care of the bulk of debugging, but sometimes an extra command can help.

5.2

5.2 Copyright Information

The following sections of this chapter fall under the copyright below: Background and Description, Syntax and Structure, and Commands.

Copyright © 1987 Regents of the University of California

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies. The University of California makes no representations about the suitability of this software for any purpose. It is provided “as is” without express or implied warranty.

5.3 Background and Description

The Tool Command Language is abbreviated as Tcl and is pronounced “tickle”. It was developed and written by Professor John Ousterhout at the University of California at Berkeley. Tcl is a combination of two main parts: a language and a library.

Language The Tcl language is a textual language intended primarily for issuing commands to interactive programs such as text editors, illustrators, shells, and most importantly debuggers. It has a set syntax and is programmable, thus allowing users to create more powerful commands than the built-in command set listed in “Swat Reference,” Chapter 4.



Library Tcl also includes a library which can be imbedded in an application, as it is in Swat. This library includes a parser for the Tcl language, routines to implement the Tcl built-in commands, and procedures allowing an application to extend Tcl with additional commands.

5.4 Syntax and Structure

5.4

Tcl supports only one type of data: *strings*. All commands, all arguments to commands, all command results, and all variable values are strings. Where commands require numeric arguments or return numeric results, the arguments and results are passed as strings. Many commands expect their string arguments to have certain formats, but this interpretation is up to the individual commands. For example, arguments often contain Tcl command strings, which may get executed as part of the commands. The easiest way to understand the Tcl interpreter is to remember that everything is just an operation on a string. In many cases Tcl constructs will look similar to more structured constructs from other languages. However, the Tcl constructs are not structured at all; they are just strings of characters, and this gives them a different behavior than the structures they may look like.

Although the exact interpretation of a Tcl string depends on who is doing the interpretation, there are three common forms that strings take: *commands*, *expressions*, and *lists*. This section will have the following main parts:

- ◆ **Basic Command Syntax**
Description of the syntax common to all Tcl code: comments, argument grouping, command grouping, variable substitution, backslash substitution.
- ◆ **Expressions**
Details on interpretation of expressions by Tcl.
- ◆ **Lists**
Details on interpretation of lists by Tcl.
- ◆ **Command Results**
What type of results a command can return.
- ◆ **Procedures**
The structure and building of procedures in Tcl.



- ◆ Variables
Variable declaration and description.

5.4.1 Basic Command Syntax

5.4

The Tcl language has syntactic similarities to both Unix and Lisp. However, the interpretation of commands is different in Tcl than in either of those other two systems. A Tcl command string consists of one or more commands separated by newline characters. Each command consists of a collection of fields separated by white space (spaces or tabs). The first field must be the name of a command, and the additional fields, if any, are arguments that will be passed to that command. For example, the command:

```
var a 22
```

has three fields: the first, **var**, is the name of a Tcl command, and the last two, **a** and **22**, will be passed as arguments to the **var** command. The command name may refer to a built-in Tcl command, an application specific command, or a command procedure defined with the built-in **proc** command. Arguments are passed literally as text strings. Individual commands may interpret those strings in any fashion they wish. The **var** command, for example, will treat its first argument as the name of a variable and its second argument as a string value to assign to that variable. For other commands, arguments may be interpreted as integers, lists, file names, or Tcl commands.

5.4.1.1 Comments

If the first non-blank character in a command is # (a number sign), then everything from the # up through the next newline character is treated as comment and discarded by the parser.

5.4.1.2 Argument Grouping

Normally each argument field ends at the next white space (tabs or spaces), but curly braces (“{” and “}”) may be used to group arguments in different ways. If an argument field begins with a left brace, then the argument is not terminated by white space; it ends at the matching right brace. Tcl will strip



off the outermost layer of braces before passing the argument to the command. For example, in the command:

```
var a {b c}
```

the **var** command will receive two arguments: **a** and **b c**. The matching right brace need not be on the same line as the left brace; in this case the newline will be included in the argument field along with any other characters up to the matching right brace. In many cases an argument field to one command consists of a Tcl command string that will be executed later; braces allow complex command structures to be built up without confusion. For example, the **eval** command takes one argument, which is a command string; **eval** invokes the Tcl interpreter to execute the command string. The command:

5.4

```
eval {  
    var a 22  
    var b 33  
}
```

will assign the value 22 to **a** and 33 to **b**.

Tcl braces act like quote characters in most other languages, in that they prevent any special interpretation of the characters between the left brace and the matching right brace.

When an argument is in braces, then command, variable, and backslash substitutions do not occur in the normal fashion; all Tcl does is to strip off the outer layer of braces and pass the contents to the command. Braces are only significant in a command field if the first character of the field is a left brace. Otherwise neither left nor right braces in the field will be treated specially (except as part of variable substitution).

5.4.1.3 Command Grouping

Normally, each command occupies one line (the command is terminated by a newline character). Thus, the string:

```
var a 22  
var b 33
```

will be interpreted as two separate commands. However, brackets may be used to group commands in ways other than one-command-per-line. If the



first character of a command is an open bracket, then the command is not terminated by a newline character; instead, it consists of all the characters up to the matching close bracket. Newline characters inside a bracketed command are treated as white space (they will act as argument separators for arguments that are not enclosed in braces). For example, the string:

```
[var a
 22] [var b 33]
```

5.4 will have the same effect as the previous example.

5.4.1.4 Command Substitution

If an open bracket occurs in any of the fields of a command, then command substitution occurs. All of the text up to the matching close bracket is treated as a Tcl command and executed immediately. The result of that command is substituted for the bracketed text. For example, consider the command:

```
var a [var b]
```

When the **var** command has only a single argument, it is the name of a variable and **var** returns the contents of that variable. In this case, if variable **b** has the value *test*, then the command above is equivalent to the command:

```
var a test
```

Brackets can be used in more complex ways. for example, if the variable **b** has the value *tmp* and the variable **c** has the value *val*, then the command:

```
var a test[var b].[var c]
```

is equivalent to the command:

```
var a testtmp.val
```

If a field is enclosed in braces then the brackets and the characters between them are not interpreted specially; they are passed through to the argument verbatim.

5.4.1.5 Variable Substitution

The dollar sign (\$) may be used as a special shorthand form for substituting variables. If \$ appears in an argument that is not enclosed in braces then



variable substitution will occur. The characters after the \$, up to the first character that is not a number, letter, or underscore, are taken as a variable name and the string value of that variable is substituted for the name. Or, if the dollar sign is followed by an open curly brace, then the variable name consists of all the characters up to the next close curly brace. For example, if variable **outfile** has the value *test*, then the command:

```
var a $outfile.c
```

is equivalent to the command:

```
var a test.c
```

and the command:

```
var a abc${outfile}tmp
```

is equivalent to the command:

```
var a abctesttmp
```

Variable substitution does not occur in arguments that are enclosed in braces: the dollar sign and variable name are passed through to the argument verbatim.

The dollar sign abbreviation is simply a shorthand form. **\$a** is completely equivalent to **[var a]**; it is provided as a convenience to reduce typing.

5.4.1.6 Backslash Substitution

Backslashes may be used to insert non-printing characters into command fields and also to insert braces, brackets, and dollar signs into fields without them being interpreted specially as previously described. The backslash sequences understood by the Tcl interpreter are listed in Table 5-1. In each case, the backslash sequence is replaced by the given character.

For example, in the command:

```
var a \{x\[ \ yz\141
```

the second argument to **var** is **{x[yza** (note the <space> as part of the argument).

If a backslash is followed by something other than one of the options in Table 5-1, then the backslash is transmitted to the argument field without



any special processing, and the Tcl scanner continues normal processing with the next character. For example, in the command:

```
var \*a \{\{test
```

the first argument will be `*a` and the second `\{test`.

5.4

Table 5-1 *Backslash Sequences*

Sequence	Replaced Value
<code>\b</code>	Backspace (octal 10)
<code>\e</code>	Escape (octal 33)
<code>\n</code>	Newline (octal 15)
<code>\t</code>	Tab (octal 11)
<code>\{</code>	Left brace (“{”)
<code>\}</code>	Right brace (“}”)
<code>\[</code>	Open bracket (“[”)
<code>\]</code>	Close bracket (“]”)
<code>\<space></code>	Space (note: does not terminate the argument)
<code>\\</code>	Backslash (“\”)
<code>\Cx</code>	Control- <i>x</i> for any ASCII <i>x</i> except M (see below)
<code>\Mx</code>	Meta- <i>x</i> for any ASCII <i>x</i>
<code>\CMx</code>	Control-meta- <i>x</i> for any ASCII <i>x</i>
<code>\ddd</code>	The digits <i>ddd</i> (one, two, or three of them) give the octal value of the character

If an argument is enclosed in braces, then backslash sequences inside the argument are parsed but no substitution occurs. In particular, backslashed braces are not counted in locating the matching right brace that terminates the argument. for example, in the command:

```
var a {\{abc}
```

the second argument to `var` will be `\{abc`.

The backslash mechanism is not sufficient to generate any argument structure; it only covers the most common cases. To produce particularly complicated arguments it will probably be easiest to use the **format** command along with command substitution.



5.4.2 Expressions

The second major interpretation applied to strings in Tcl is as *expressions*. Several commands, such as **expr**, **for**, and **if**, treat some of their arguments as expressions and call the Tcl expression processor (Tcl_Expr) to evaluate them. A Tcl expression has C-like syntax and evaluates to an integer result. Expressions may contain integer values, variable names in \$ notation (the variables' values must be integer strings), commands (embedded in brackets) that produce integer string results, parentheses for grouping, and operators. Numeric values, whether they are passed directly or through variable or command substitution, may be specified either in decimal (the normal case), in octal (if the first character of the value of the first character is 0 (zero)), or in hexadecimal (if the first two characters of the value are 0x). The valid operators are listed in Table 5-2 grouped in decreasing order of precedence.

5.4

Table 5-2 *Valid Operators*

Operators	Description
- ~ !	Unary minus, bit-wise NOT, logical NOT
* / %	Multiply, divide, remainder
+ -	Add and subtract
<< >>	Left and right shift
< > <= >=	Boolean less, greater, less than or equal, and greater than or equal. Each operator produces 1 if the condition is true, 0 otherwise
== !=	Boolean equal and not equal
&	Bit-wise AND
^	Bit-wise exclusive OR
	Bit-wise OR
&&	Logical AND
	Logical OR

See a C manual for more details on the results produced by each operator. All of the binary operators group left to right within the same precedence level. for example, the expression:

```
( 4 * 2 ) < 7
```

evaluates to zero. Evaluating the expression string:

Tools 

```
( $\$a+3$ )<[var b]
```

will cause the values of the variables **a** and **b** to be examined; the result will be 1 if **b** is greater than **a** by at least 3; otherwise the result will be 0.

In general it is safest to enclose an expression in braces when entering it in a command; otherwise, if the expression contains any white space then the Tcl interpreter will split it among several arguments. For example, the command:

5.4

```
expr  $\$a + \$b$ 
```

results in three arguments being passed to **expr**: **\$a**, **+**, and **\$b**. In addition, if the expression is not in braces then the Tcl interpreter will perform variable and command substitution immediately (it will happen in the command parser rather than in the expression parser). In many cases the expression is being passed to a command that will evaluate the expression later (or even many times if, for example, the expression is to be used to decide when to exit a loop). usually the desired goal is to re-do the variable or command substitutions each time the expression is evaluated, rather than once and for all at the beginning. For an example of a mistake, the command:

```
for {var i 1}  $\$i \leq 10$  {var i [expr  $\$i+1$ ]} {body...}
```

is probably intended to iterate over all values of **i** from 1 to 10. After each iteration of the body of the loop, for will pass its second argument to the expression evaluator to see whether or not to continue processing. Unfortunately, in this case the value of **i** in the second argument will be substituted once and for all when the for command is parsed. If **i** was 0 before the for command was invoked then **for**'s second argument will be **0<=10** which will always evaluate to 1, even though **i**'s value eventually becomes greater than 10. In the above case the loop will never terminate. By placing the expression in braces, the substitution of **i**'s value will be delayed; it will be re-done each time the expression is evaluated, which is probably the desired result:

```
for {var i 1} { $\$i \leq 10$ } {var i [expr  $\$i+1$ ]} {body...}
```



5.4.3 Lists

The third major way that strings are interpreted in Tcl is a *list*. A list is just a string with a list-like structure consisting of fields separated by white space. For example, the string:

```
Al Sue Anne John
```

is a list with four elements or fields. Lists have the same basic structure as command strings, except that a newline character in a list is treated as a field separator just like a space or tab. Conventions for braces and backslashes are the same for lists as for commands. For example, the string:

5.4

```
a b\ c {d e {f g h}}
```

is a list with three elements: **a**, **b c**, and **d e {f g h}**. Note the space between the **b** and **c**. Whenever an element is extracted from a list, the same rules about backslashes and braces are applied as for commands. Thus in the above example when the third element is extracted from the list, the result is:

```
d e {f g h}
```

(when the field was extracted, all that happened was to strip off the outermost layer of braces). Command substitution is never made on a list (at least, not by the list-processing commands; the list can always be passed to the Tcl interpreter for evaluation).

The Tcl commands **concat**, **foreach**, **index**, **length**, **list**, and **range** allow you to build lists, extract elements from them, search them, and perform other list-related functions.

5.4.4 Command Results

Each command produces two results: a *code* and a *string*. The code indicates whether the command completed successfully or not, and the string gives additional information. The valid codes are defined as follows:

TCL_OK	This is the normal return code, and indicates that the command completed successfully. The string gives the command's return value.
--------	---



TCL_ERROR

Indicates that an error occurred; the string gives a message describing the error.

TCL_RETURN

Indicates that the return command has been invoked, and that the current procedure should return immediately. The string gives the return value that procedure should return.

5.4

TCL_BREAK

Indicates that the break command has been invoked, so the innermost loop should abort immediately. The string should always be empty.

TCL_CONTINUE

Indicates that the continue command has been invoked, so the innermost loop should go on to the next iteration. The string should always be empty.

Tcl programmers do not normally need to think about return codes, since **TCL_OK** is almost always returned. If anything else is returned by a command, then the Tcl interpreter immediately stops processing commands and returns to its caller. If there are several nested invocations of the Tcl interpreter in progress, then each nested command will usually return the error to its caller, until eventually the error is reported to the top-level application code. The application will then display the error message for the user.

In a few cases, some commands will handle certain “error” conditions themselves and not return them upwards. For example, the **for** command checks for the **TCL_BREAK** code; if it occurs, then **for** stops executing the body of the loop and returns **TCL_OK** to its caller. The **for** command also handles **TCL_CONTINUE** codes and the procedure interpreter handles **TCL_RETURN** codes. The **catch** command allows Tcl programs to catch errors and handle them without aborting command interpretation any further.

5.4.5 Procedures

Tcl allows one to extend the command interface by defining procedures. A Tcl procedure can be invoked just like any other Tcl command (it has a name and



it receives one or more arguments). The only difference is that its body is not a piece of C code linked into the program; it is a string containing one or more other Tcl commands. See the **proc** command for information on how to define procedures and what happens when they are invoked.

5.4.6 Variables

Tcl allows the definition of variables and the use of their values either through \$-style variable substitution, the **var** command, or a few other mechanisms. Variables need not be declared: a new variable will automatically be created each time a new variable name is used. Variables may be either global or local. If a variable name is used when a procedure is not being executed, then it automatically refers to a global invocation of the procedure. Local variables are deleted whenever a procedure exits. The **global** command may be used to request that a name refer to a global variable for the duration of the current procedure (somewhat analogous to **extern** in C).

5.5

5.5 Commands

The Tcl library provides the following built-in commands, which will be available to any application using Tcl. In addition to these built-in commands, there may be additional commands defined in Swat, plus commands defined as Tcl procedures.

5.5.1 Notation

The descriptions of the Tcl commands will follow the following notational conventions:

- ◆ **command (alternative1 | alternative2 | ... | alternativeN)**
() The parentheses enclose a set of alternatives separated by a vertical line. For example, the expression **quit (cont | leave)** means that either **quit cont** or **quit leave** can be used.



- ◆ **command [optional_argument]**
[] The brackets enclose optional arguments to the command. For example, the command **alias** [**<word[<command>]>**] could have zero, one, or two arguments because the *<command>* and *<word>* arguments are optional. Another example would be the command **objwalk** [**<addr>**], which may take zero arguments if it is meant to use the default address, and one argument if the user gives it a particular address to look at.
- 5.5 ◆ **command <type_of_argument>**
< > The angled brackets enclose the type of an argument rather than the actual string to be typed. For example, **<addr>** indicates an address expression and **<argument>** indicates some sort of argument, but **(addr|type)** means either the string **addr** or the string **type**.
- ◆ * +
An asterisk following any of the previous constructs indicates zero or more repetitions of the construct may be typed. An addition sign indicates one or more repetitions of the construct may be used. For example, **unalias word *** can be the **unalias** command by itself, or it can be followed by a list of words to be unaliased.

5.5.2 Built-in Commands

The built-in Tcl commands are as follows:

■ bc

Usage:

- bc list <proc>
- bc disasm <proc>
- bc compile <proc>
- bc fcompile <file> [<nohelp>]
- bc fload <file>
- bc fdisasm <file>
- bc debug [1|0]

Examples:

“bc compile poof”

Compiles the body of the procedure “poof” and replaces the existing procedure with its compiled form.



`"bc fcomp bptutils.tcl"`

Creates the file `"bptutils.tlc"` that contains a stream of compiled Tcl that will do exactly what sourcing `bptutils.tcl` does, except the resulting procedures will be compiled Tcl, not interpreted Tcl.

`"bc fload bptutils.tlc"`

Loads a file containing a stream of compiled Tcl code.

Synopsis: The `"bc"` command allows you to create and examine compiled Tcl code. Compiled Tcl is not nearly as readable or changeable as interpreted Tcl code, but it's 30-50% faster.

5.5

Notes: The `"list"` subcommand doesn't work as yet. Eventually it will attempt to construct a more readable form of compiled code. For now, the raw opcodes will have to do.

See Also: **source**.

■ break

Usage: `break`

Examples:

`"break"` Break out of the current loop.

Synopsis: Breaks out of the current loop or the current nested interpreter.

Notes:

- ◆ Only the closest-enclosing loop can be exited via this command.
- ◆ This command may be invoked only inside the body of a loop command such as **for** or **foreach**. It returns a `TCL_BREAK` code to signal the innermost containing loop command to return immediately.
- ◆ If you've entered a nested interpreter, e.g. by calling a function in the patient, use this to exit the interpreter and restore the registers to what they were before you made the call.

See Also: `continue`, `for`.

■ case

Usage: `case <string> [in] [<pat> <body>]+`

Tool Command Language

280

Examples:

5.5

```
"[case $c in
    {[0-9]} {
        # do something with digit
    }
    default {
        # do something with non-digit
    }
]"
```

Do one of two things depending on whether the character in `$c` is a digit.

Synopsis:

Perform one of a set of actions based on whether a string matches one or more patterns.

Notes:

- ◆ Compares each of the *<pattern>* arguments to the given *<string>*, executing *<body>* following the first *<pattern>* to match. *<pattern>* uses shell wildcard characters as for the string match command, but may also contain alternatives, which are separated by a vertical bar, thus allowing a *<body>* to be executed under one of several circumstances. In addition, if one *<pattern>* (or element thereof) is the string default, the associated *<body>* will be executed if none of the other patterns matches. For example, the following:

```
[case $test in
    a|b {return 1}
    {default|[DE]a*} {return 0}
    ?c {return -1}]
```

will return 1 if variable **test** contains **a** or **b**, -1 if it contains a two-letter string whose second letter is **c**, and 0 in all other cases, including the ones where **test**'s first two letters are either **Da** or **Ea**.

- ◆ Each *<pat>* argument is a list of patterns of the form described for the "string match" command.
- ◆ Each *<pat>* argument must be accompanied by a *<body>* to execute.



- ◆ If a <pat> contains the special pattern “default,” the associated <body> will be executed if no other pattern matches. The difference between “default” and “*” is a pattern of “*” causes the <body> to be executed regardless of the patterns in the remaining <pat> arguments, while “default” postpones the decision until all the remaining patterns have been checked.
- ◆ You can give the literal “in” argument if you wish to enhance the readability of your code.

See Also: string, if.

5.5

■ catch

```
catch <command> [<varName>]
```

Synopsis: Executes a command, retaining control even if the command generates an error (which would otherwise cause execution to unwind completely).

Notes:

- ◆ The **catch** command may be used to prevent errors from aborting command interpretation. **catch** calls the Tcl interpreter recursively to execute <command>, and always returns a TCL_OK code, regardless of any errors that might occur while executing <command>. The return value from **catch** is a decimal string giving the code returned by the Tcl interpreter after executing <command>. This will be zero (TCL_OK) if there were no errors in command; otherwise it will have a non-zero value corresponding to one of the exceptional return codes. If the <varName> argument is given, then it gives the name of a variable; **catch** will set the value of the variable to the string returned from command (either a result or an error message).
- ◆ This returns an integer that indicates how <command> completed:
 - 0** Completed successfully; \$<varName> contains the result of the command.
 - 1** Generated an error; \$<varName> contains the error message.
 - 2** Executed “return”; \$<varName> contains the argument passed to “return.”
 - 3** Executed “break”; \$<varName> is empty.
 - 4** Executed “continue”; \$<varName> is empty.

Tool Command Language

282

See Also: protect.

■ concat

Usage: concat <arg1>+

Examples:

“concat \$list1 \$list2”

Merges the lists in \$list1 and \$list2 into a single list whose elements are the elements of the two lists.

Synopsis: Concatenates multiple list arguments into a single list.

Notes:

- ◆ This command treats each argument as a list and concatenates them into a single list. It permits any number of arguments. For example, the command

```
concat a b {c d e} {f {g h}}
```

will return **a b c d e f {g h}** as its result.

- ◆ There is a sometimes-subtle difference between this in the “list” command: Given two lists, “concat” will form a list whose n elements are the combined elements of the two component lists, while “list” will form a list whose 2 elements are the two lists. For example,

```
concat a b {c d e} {f {g h}}
```

yields the list

```
a b c d e f {g h}
```

but

```
list a b {c d e} {f {g h}}
```

yields

```
a b {c d e} {f {g h}}
```

See Also: list.

■ continue

Usage: continue

Examples:



“continue” Return to the top of the enclosing loop.

Synopsis: Skips the rest of the commands in the current loop iteration, continuing at the top of the loop.

Notes:

- ◆ Only the closest-enclosing loop can be continued via this command.
- ◆ The <next> clause of the “for” command is not part of the current iteration, i.e. it will be executed even if you execute this command.
- ◆ This command may be invoked only inside the body of a loop command such as **for** or **foreach**. It returns a TCL_CONTINUE code to signal the innermost containing loop command to skip the remainder of the loop’s body but continue with the next iteration of the loop.

5.5

See Also: break, for.

■ defsubr

Usage: defsubr <name> <args> <body>

Examples:

“defsubr poof {arg1 args} {return [list \$arg1 \$args]}”
 Defines a procedure poof that takes 1 or more arguments and merges them into a list of two elements.

Synopsis: This is the same as the “proc” command, except the new procedure’s name may not be abbreviated when it is invoked.

Notes: Refer to the documentation for **proc** for more information.

■ error

Usage: error <message>

Examples:

“error {invalid argument}”
 Generates an error, giving the not-so-helpful message “invalid argument” to the caller’s caller.

Notes:

- ◆ Unless one of the procedures in the call stack has executed a “catch” command, all procedures on the stack will be terminated with <message>

(and an indication of an error) being the result of the final one so terminated.

- ◆ Any commands protected by the “protect” command will be executed.

See Also: return, catch.

■ eval

5.5

Usage: eval <body>

Examples:

“eval \$mangled_command”

Evaluate the command contained in \$mangled_command and return its result.

Synopsis: Evaluates the passed string as a command and returns the result of that evaluation.

- ◆ **eval** takes one argument, which is a Tcl command (or collection of Tcl commands separated by newlines in the usual way). **eval** evaluates <body> by passing it to the Tcl interpreter recursively, and returns the result of the last command. If an error occurs inside <body> then **eval** returns that error.
- ◆ This command is useful when one needs to cobble together a command from arguments or what have you. For example, if one of your arguments is a list of arguments to pass to another command, the only way to accomplish that is to say something like “eval [concat random-command \$args]”, which will form a list whose first element is the command to be executed, and whose remaining elements are the arguments for the command. “eval” will then execute that list properly.
- ◆ If the executed command generates an error, “eval” will propagate that error just like any other command.

See Also: concat, list.

■ expr

Usage: expr <expression> [float]

Examples:

“expr 36*25” Multiplies 36 by 25 and returns the result.



“expr \$i/6 float”

Divides the number in \$i by 6 using floating-point arithmetic; the result is a real number.

“expr 7.2*10 float”

Multiplies 7.2 by 10. Note that though the answer (72) is an integer, we need to pass the “float” keyword to make sure that the expression is interpreted correctly.

Synopsis: Evaluates an arithmetic expression and returns its value.

5.5

Notes:

- ◆ Most C operators are supported with the standard operator precedence.
- ◆ If you use a Tcl variable in the expression, the variable may only contain a number; it may not contain an expression.
- ◆ The result of any Tcl command, in square brackets (“[]”) must be a number; it may not be an expression.
- ◆ All the C and Esp radix specifiers are allowed.
- ◆ Bitwise and boolean operators (!, &, ^, |, &&, ||, >>, <<, ~) are not permitted when the expression is being evaluated using floating-point arithmetic.

■ file

Usage:

```
file dirname <name>
file exists <name>
file extension <name>
file isdirectory <name>
file isfile <name>
file readable <name>
file rootname <name>
file tail <name>
file writable <name>
file match <pattern>
file newer <name1> <name2>
```

Examples:

“file match /pcgeos/tcl/*.tcl”

Looks for all files/directories in /pcgeos/tcl whose name ends with “.tcl”.

“file isdir \$path”

See if the path stored in \$path refers to a directory.

“file tail \$path”

Return the final component of the path stored in \$path

5.5 **Synopsis:** Performs various checks and manipulations of file and directory names.

Notes:

- ◆ The forward slash is the path separator for this command.
- ◆ The predicate subcommands (executable, exists, isdirectory, isfile, owned, readable, and writable) all return 1 if the path meets the requirements, or 0 if it doesn't.
- ◆ “file match” takes a *pattern* made from the same components as are described for “string match”. It is *not* the same as the standard DOS wildcarding, where ‘.’ serves to separate the root pattern from the extension pattern. For this command “*.*” would match only files that actually have an extension.
- ◆ “file dirname” returns the directory portion of *name*. If *name* has no directory portion, this returns “.”
- ◆ “file rootname” returns all leading directory components of *name*, plus the text before its extension, without the “.” that separates the name from the extension.
- ◆ “file tail” returns all of the characters in *name* after the final forward slash, or *name* if it contains no forward slashes.
- ◆ “file newer” returns 1 if *name1* was modified after *name2*. It returns 0 otherwise.

See Also: string.

■ **for**

Usage: for <start> <test> <next> <body>

Examples:



```
"for {var i 0} {$i < 10} {var i [expr $i+1]} {echo $i}"
Prints the numbers from 0 to 9.
```

Synopsis: This is Tcl's main looping construct. It functions similarly to the "for" in C.

Notes:

- ◆ <start> is a Tcl command string (which may involve multiple commands over multiple lines, if desired) that is executed once at the very start of the loop. It is always executed. If it returns an error, or contains a "break" command, no part of the loop will execute.
- ◆ <test> is an arithmetic expression that is passed to the "expr" command. If the result is non-zero, the <body> is executed.
- ◆ <next> is a Tcl command string (which may involve multiple commands over multiple lines, if desired) that is executed at the end of each iteration before <test> is evaluated again. If it returns an error, or contains a "break" command, no part of the loop will execute.
- ◆ You can exit the loop prematurely by executing the "break" command in any of the three Tcl command strings (<start>, <next>, or <body>).
- ◆ So long as there's no error, "for" always returns the empty string as its result.
- ◆ If a **continue** command is invoked within <body> then any remaining commands in the current execution of <body> are skipped; processing continues by invoking the Tcl interpreter on <next>, then evaluating <test>, and so on. If a **break** command is invoked within <body>, then the **for** command will return immediately. The operation of **break** and **continue** are similar to the corresponding statements in C.

5.5

See Also: foreach, break, continue.

■ foreach

Usage: foreach <varname> <list> <body>

Examples:

```
"foreach el $list {echo poof = $el}"
Prints each element of the list $list preceded by the profound
words "poof = "
```

Synopsis: This is a looping construct to easily iterate over all the elements of a list.



Notes:

- ◆ `<body>` is evaluated once for each element in `<list>`. Before each evaluation, the next element is placed in the variable `<varName>`.
- ◆ You can exit the loop prematurely by executing the “break” command.
- ◆ As long as there’s no error, “foreach” always returns the empty string.
- ◆ The **break** and **continue** statements may be invoked inside `<body>`, with the same effect as in the **for** command.

5.5

■ format

```
format <formatString> [<arg> ]*
```

This command generates a formatted string in the same way as the C **sprintf** procedure (it uses **sprintf** in its implementation). `<formatString>` indicates how to format the result, using % fields as in **sprintf**, and the additional arguments, if any, provide values to be substituted into the result. All of the **sprintf** options are valid; see the **sprintf** procedure in a C manual for more details. Each `<arg>` must match the expected type from the % field in `<formatString>`; the **format** command converts each argument to the correct type (floating, integer, etc.) before passing it to **sprintf** for formatting. The only unusual conversion is for `%c`; in this case the argument must be a decimal string, which will then be converted to the corresponding ASCII character value. **format** does backslash substitution on its `<formatString>` argument, so backslash sequences in `<formatString>` will be handled correctly even if the argument is in braces. The return value from **format** is the formatted string.

■ global

Usage: global <varname>+

Examples:

“global attached”

When next the “attached” variable is fetched or set, get it from the global scope, not the local one.

Synopsis: Declares the given variables to be from the global scope.

Notes:



- ◆ For the duration of the procedure in which this command is executed (but not in any procedure it invokes), the global variable of the given name will be used when the variable is fetched or set.
- ◆ If no global variable of the given name exists, the setting of that variable will define it in the global scope.
- ◆ This command is ignored unless a Tcl procedure is being interpreted. If so, then it declares the given *<varname>*'s to be global variables rather than local ones. For the duration of the current procedure (and only while executing in the current procedure), any reference to any of the *<varname>* values will be bound to a global variable instead of a local one.

5.5

See Also: `var`.

■ if

Usage: `if <test> [then] <trueBody>`
 `(elif <test> [(then)] <trueBody>)*`
 `[[else] <falseBody>]`

Examples:

`"if {$v > 3} {echo yes} {echo no}"`
 Prints "yes" if \$v is greater than 3, else it prints "no".

`"if {$v > 3} then {echo yes} else {echo no}"`
 Ditto.

`"if {$v > 3} then {echo yes} elif {$v == 3} {echo maybe} else {echo no}"`

Synopsis: This is Tcl's conditional, as you'd expect from its name.

Notes:

- ◆ The "then" and "else" keywords are optional, intended to delineate the different sections of the command and make the whole easier to read.
- ◆ The "elif" keyword is *mandatory* if you want to perform additional tests.
- ◆ The *<expr>* arguments are normal Tcl expressions. If the result is non-zero, the appropriate *<truebody>* is executed. If none of the *<expr>* arguments evaluates non-zero, *<falsebody>* is executed.
- ◆ If a *<truebody>* is empty and the test evaluated non-zero, "if" will return the result of the test. Otherwise "if" returns the result from last

command executed in whichever *<truebody>* or *<falsebody>* argument was finally executed. It returns an empty string if no *<expr>* evaluated non-zero and no *<falsebody>* was given.

5.5

- ◆ The **if** command evaluates *<test>* as an expression in the same way that **expr** evaluates its argument. If the result is non-zero then *<trueBody>* is called by passing it to the Tcl interpreter. Otherwise *<falseBody>* is executed by passing it to the Tcl interpreter. *<falseBody>* is also optional; if it isn't specified then the command does nothing if *<test>* evaluates to zero. The return value from **if** is the value of the last command executed in *<trueBody>* or *<falseBody>* or the empty string if *<test>* evaluates to zero and *<falseBody>* isn't specified. Alternative test conditions can be added by adding *<elif>* arguments.

See Also: *expr*.

■ index

Usage: *index <value> <index> [chars]*

Examples:

"index {a b c} 1"

Extracts "b" from the list.

"index {hi mom} 3 char"

Extracts "m" from the string.

Synopsis: *"index"* is used to retrieve a single element or character from a list or string.

Notes:

- ◆ Elements and characters are numbered from 0.
- ◆ If you request an element or character from beyond the end of the *<list>* or *<string>*, you'll receive an empty list or string as a result.
- ◆ If the *<chars>* keyword isn't specified, then **index** treats *<value>* as a list and returns the *<index>*'th field from it. In extracting the field, **index** observes the same rules concerning braces and backslashes as the Tcl command interpreter; however, variable substitution and command substitution do not occur. If the *<chars>* keyword is specified (or any abbreviation of it), then *<value>* is treated as a string and the command returns the *<index>*'th character from it (or the empty string if there aren't at least *<index>*+1 characters in the string).



■ info

Usage:

```

info args <procname> [<pattern>]
info arglist <procname>
info body <procname>
info cmdcount
info commands [<pattern>]
info default <procname> <arg> <varname>
info globals [<pattern>]
info locals [<pattern>]
info procs [<pattern>]
info vars [<pattern>]

```

5.5

Examples:

“info args fmtval” Retrieves the names of the arguments for the “fmtval” command so you know in what order to pass things.

“info body print-frame” Retrieves the string that is the body of the “print-frame” Tcl procedure.

“info commands *reg*” Retrieves a list of commands whose names contain the string “reg”.

Synopsis: This command provides information about a number of data structures maintained by the Tcl interpreter.

Notes:

- ◆ All the <pattern> arguments are standard wildcard patterns as are used for the “string match” and “case” commands. See “string” for a description of these patterns.
- ◆ “info args” returns the complete list of arguments for a Tcl procedure, or only those matching the <pattern>, if one is given. The arguments are returned in the order in which they must be passed to the procedure.
- ◆ “info arglist” returns the complete list of arguments, and their default values, for a Tcl procedure.
- ◆ “info body” returns the command string that is the body of the given Tcl procedure.
- ◆ “info cmdcount” returns the total number of commands the Tcl interpreter has executed in its lifetime.

5.5

- ◆ “info commands” returns the list of all known commands, either built-in or as Tcl procedures, known to the interpreter. You may also specify a pattern to restrict the commands to those whose names match the pattern.
- ◆ “info default” returns non-zero if the argument named <arg> for the given Tcl procedure has a default value. If it does, that default value is stored in the variable whose name is <varname>.
- ◆ “info globals” returns the list of all global variables accessible within the current variable scope (i.e. only those that have been declared global with the “global” command, unless you issue this command from the command-line, which is at the global scope), or those that match the given pattern.
- ◆ “info locals” returns the list of all local variables, or those that match the given pattern.
- ◆ “info procs” returns the list of all known Tcl procedures, or those that match the given pattern.
- ◆ “info vars” returns the list of all known Tcl variables in the current scope, either local or global. You may also give a pattern to restrict the list to only those that match.

See Also: proc, defcmd, defcommand, defsubr.

■ length

Usage: length <value> [<chars>]

Examples:

“length \$args”

Returns the number of elements in the list \$args

“length \$str char”

Returns the number of characters in the string \$str

Synopsis: Determines the number of characters in a string, or elements in a list.

Notes: If (*chars*) isn't specified, **length** treats <value> as a list and returns the number of elements in the list. If <chars> is specified (or any abbreviation of it), then **length** treats <value> as a string and returns the number of characters in it (not including the terminating null character).

See Also: index, range.



list

Usage: list <arg>+

Examples:

“list a b {c d e} {f {g h}}”

Returns the list “a b {c d e} {f {g h}}”

Synopsis: Joins any number of arguments into a single list, applying quoting braces and backslashes as necessary to form a valid Tcl list.

5.5

Notes:

- ◆ If you use the “index” command on the result, the 0th element will be the first argument that was passed, the 1st element will be the second argument that was passed, etc.
- ◆ The difference between “list” and “concat” is subtle. Given the above arguments, “concat” would return “a b c d e f {g h}”.
- ◆ This command returns a list comprised of all the <args>. It also adds braces and backslashes as necessary, so that the **index** command may be used on the result to re-extract the original arguments, and also so that **eval** may be used to execute the resulting list, with <arg1> comprising the command's name and the other <args> comprising its arguments.

See Also: concat, index, range.

proc

Usage: proc <name> <args> <body>

Examples:

“proc poof {{arg1 one} args} {return [list \$arg1 \$args]}”

Defines a procedure poof that takes 0 or more arguments and merges them into a list of two elements. If no argument is given, the result will be the list {one {}}

Synopsis: Defines a new Tcl procedure that can be invoked by typing a unique abbreviation of the procedure name.

Notes:

- ◆ Any existing procedure or built-in command with the same name is overridden.

- ◆ `<name>` is the name of the new procedure and can consist of pretty much any character (even a space or tab, if you enclose the argument in braces).
- ◆ `<args>` is the, possibly empty, list of formal parameters the procedure accepts. Each element of the list can be either the name of local variable, to which the corresponding actual parameter is assigned before the first command of the procedure is executed, or a two-element list, the first element of which is the local variable name, as above, and the second element of which is the value to assign the variable if no actual parameter is given.
- ◆ If the final formal parameter is named “args”, the remaining actual parameters from that position on are cobbled into a list and assigned to the local variable `$args`. This allows a procedure to receive a variable number of arguments (even 0, in which case `$args` will be the empty list).
- ◆ If the only formal parameter is “noeval”, all the actual parameters are merged into a list and assigned to `$noeval`. Moreover, neither command- nor variable-substitution is performed on the actual parameters.
- ◆ The return value for the procedure is specified by executing the “return” command within the procedure. If no “return” command is executed, the return value for the procedure is the empty string.
- ◆ Whenever the new command is invoked, the contents of `<body>` will be executed by the Tcl interpreter. `<args>` specifies the formal arguments to the procedure. It consists of a list, possibly empty, each of whose elements specifies one argument. Braces and backslashes may be used in the usual way to specify complex default values.
- ◆ When `<name>` (or a unique abbreviation of same) is invoked, a local variable will be created for each of the formal arguments to the procedure; its value will be the value of corresponding argument in the invoking command or the argument's default value. Arguments with default values need not be specified in a procedure invocation. However, there must be enough actual arguments for all the formal arguments that don't have defaults, and there must not be any extra actual arguments (unless the “args” keyword was used).
- ◆ When `<body>` is being executed, variable names normally refer to local variables, which are created automatically when referenced and deleted when the procedure returns. One local variable is automatically created for each of the procedure's arguments. Global variables can only be accessed by invoking the **global** command.



- ◆ The **proc** command itself returns the null string.

See Also: defsubr, return.

■ protect

Usage: protect <body> <cleanup>

Examples:

```
“protect {
    var s [stream open $file w]
    # do stuff with the stream
} {
    catch {stream close $s}
}”
```

5.5

Perform some random operations on a file making sure the stream gets closed, even if the user types control-C.

Synopsis: Allows one to ensure that clean-up for a sequence of commands will always happen, even if the user types control-C to interrupt the command.

Notes:

- ◆ Since the interrupt can come at any time during the <body>, the <cleanup> command string should not rely on any particular variables being set. Hence the “catch” command used in the <cleanup> clause of the example.
- ◆ The <cleanup> clause will also be executed if any command in the <body> generates an error.

See Also: catch.

■ range

Usage: range <value> <first> <last> [chars]

Examples:

```
“range {a b c} 1 end”
Returns {b c} (element 1 to the end)
```

Tool Command Language

296

“range {hi mom} 3 end chars”
Returns “mom”

Synopsis: Extracts a range of characters from a string, or elements from a list.

Notes:

5.5

- ◆ If you give an ending index that is greater than the number of elements in the list (characters in the string), it will be adjusted to be the index of the last element (character).
- ◆ If you give a starting index that is greater than the number of elements in the list (characters in the string), the result will be the empty list (string).
- ◆ You can give <end> as “end” (without the quotation marks, of course) to indicate the extraction should go to the end of the list (string).
- ◆ The range is inclusive, so “range {a b c} 0 0” returns “a”.
- ◆ Neither index may be less than 0 or “range” will generate an error.
- ◆ Return a range of fields or characters from value. If the *chars* keyword, or any abbreviation of it, is specified, then **range** treats <value> as a character string and returns characters <first> through <last> of it, inclusive. If <last> is less than <first> then an empty string is returned. Note: **range value first first** does not always produce the same results as **index value first** (although it often does for simple fields that are not enclosed in braces); it does, however, produce exactly the same results as **list [index value first]**.

See Also: index.

■ return

Usage: return [<value>]

Examples:

“return \$val”
Returns the string in \$val as the value for the current Tcl procedure.

Synopsis: Causes an immediate return from the current Tcl procedure, with or without a value.

Notes:



- ◆ Every Tcl procedure returns a string for a value. If the procedure was called via command substitution (having been placed between square brackets as the argument to another command), the return value takes the place of the command invocation.
- ◆ Execution of the current procedure terminates immediately, though any <cleanup> clause for a containing “protect” command will still be executed.
- ◆ If no “return” command is invoked within a Tcl procedure, the procedure returns the empty string by default.
- ◆ This command may be invoked only when a procedure call is in progress. It causes the current procedure to return immediately. If <value> is specified, it will be the return value from the procedure. Otherwise the current procedure will return the empty string.

5.5

See Also: error, proc, defsubr, defcommand, defcmd.

■ scan

Usage: scan <string> <format> [<varname1>]*

Examples:

```
“scan $input {my name is %s} name”
    Trims the leading string “my name is “ from the string in
    $input and stores the rest of the string within the variable
    $name
```

Synopsis: “scan” parses fields from an input string, given the string and a format string that defines the various types of fields. The fields are assigned to variables within the caller’s scope.

Notes:

- ◆ The <format> string consists of literal text, which must be matched explicitly, and field definitions. The <varName> arguments are names of variables to which each successive field value is assigned.
- ◆ A single whitespace character (space or tab) will match any number of whitespace characters in the input string. Fields are specified as for the standard C library routine “scanf”:

5.5

%c A single character. The field value stored is the decimal number of the ASCII code for the character scanned. So if the character were a space, the variable would receive the string "32".

%d A signed decimal integer is parsed and stored.

%o An octal integer is parsed and stored, as a decimal number.

%x A hexadecimal integer is parsed and stored, as a decimal number.

%i A signed integer, following the standard C radix-specification standard, is parsed and stored as a decimal number.

%f A floating-point number is parsed as a "float" and stored without exponent, unless the exponent is less than -4.

%s A whitespace-terminated string is parsed and stored.

%[<char-class>]

A string consisting only of the characters in the given character class (see "string match" for details on character classes) is parsed and stored. The normal leading-whitespace skipping is suppressed.

%% Matches a single percent sign in the input.

- ◆ If the % of a field specifier is followed by an *, the field is parsed as usual, consuming characters from the string, but the result is not stored anywhere and you should not specify a variable to receive the value.
- ◆ The maximum length of a field may be specified by giving a decimal number between the % and the field-type character. So "%10s" will extract out a string of at most 10 characters.
- ◆ There is currently a limit of 5 fields.

See Also: `format`.

■ **source**

Usage: `source <fileName>`

Examples:

`"source coolness"`

Evaluates all commands within the file "coolness.tcl" in the current directory.



Synopsis: Reads and evaluates commands from a file.

Notes:

- ◆ If <file> has no extension and doesn't exist, "source" will append ".tcl" to the end and try and read that file.
- ◆ The return value of **source** is the return value of the last command executed from the file. If an error occurs in executing the contents of the file, then the **source** command will return that error.

5.5

■ string

Usage: string compare<string1> <string2> [no_case]
string first<substring> <string> [no_case]
string last<substring> <string> [no_case]
string match<string> <pattern>
string subst <string> <search> <replace> [global]

Examples:

"if {[string c [index \$args 1] all] == 0}"
Do something if the 2nd element of the list in \$args is the string "all".

"while {[string m [index \$args 0] -*]}"
Loop while the first element of the list in \$args begins with a hyphen.

Synopsis: Examine strings in various ways.

Notes:

- ◆ "string subst" searches <string> for occurrences of <search> and replaces them with <replace>. If 5th argument is given as "global" (it may be abbreviated), then all (non-overlapping) occurrences of <search> will be replaced. If 5th argument is absent, only the first occurrence will be replaced.
- ◆ "string compare" compares the two strings character-by-character. It returns -1, 0, or 1 depending on whether <string1> is lexicographically less than, equal to, or greater than <string2>. If the no_case parameter is passed then it does a case insensitive compare.
- ◆ "string first" searches <string> for the given <substring>. If it finds it, it returns the index of the first character in the first such match. If

5.5

<substring> isn't part of <string>, it returns -1. If the `no_case` parameter is passed it does the search ignoring case.

- ◆ “string last” is much like “string first”, except it returns the index of the first character of the last match for the <substring> within <string>. If there is no match, it returns -1.
- ◆ “string match” compares <string> against <pattern> and returns 1 if the two match, or 0 if they do not. For the strings to match, their contents must be identical, except that the following special sequences may appear in <pattern> with the following results:

* Matches any sequence of characters, including none.

? Matches any single character

[<char-class>]

Matches a single character within the given set. The elements of the set are specified as single characters, or as ranges of the form <start>-<end>. Thus [0-9x] matches a single character that is a numeric digit or the letter x.

[^<char-class>]

Matches a single character *not* within the given set.

* Matches an asterisk.

\? Matches a question mark.

\[Matches an open-bracket.

See Also: case.

■ uplevel

Usage: uplevel <level> <body>
uplevel <function> <body>

Examples:

“uplevel print-frame {var found1}”

Sets \$found to 1 within the variables belonging to the nearest invocation of print-frame on the call stack.

“uplevel 0 {var foo-table}”

Retrieves the value of the global variable foo-table.



“uplevel 1 {var found 1}”

Sets \$found to 1 within the scope of the procedure that called the one executing the “uplevel” command.

Synopsis: Provides access to the variables of another procedure for fairly specialized purposes.

Notes:

◆ <level> is a signed integer with the following meaning:

5.5

> 0 Indicates the number of scopes to go up. For example, if you say “uplevel 1 {var foo 36}”, you would modify (or create) the variable “foo” in your caller’s scope.

<= 0 Indicates the number of scopes to go down from the global one. “uplevel 0 <body>” will execute <body> in the top-most scope, which means that no local variables are involved, and any variables created by the commands in <body> persist as global variables.

◆ <function> is the name of a function known to be somewhere on the call stack. If the named function isn’t on the call stack anywhere, “uplevel” generates an error.

◆ <body> may be spread over multiple arguments, allowing the command to be executed to use variables local to the current procedure as arguments without having to use the “list” command to form the <body>.

See Also: global.

■ var

Usage: var <varname>
var (<name> <value>)+

Examples:

“echo [var poof]”
Prints the value stored in the variable “poof”

“var a b c d” Assigns the string “b” to the variable “a”, and the string “d” to the variable “c”.

“var yes \$no no \$yes”
Exchanges the values of the “yes” and “no” variables

Synopsis: This is the means by which variables are defined in Tcl. Less often, it is also used to retrieve the value of a variable (usually that's done via variable substitution).

Notes:

5.6

- ◆ If you give only one argument, the value of that variable will be returned. If the variable has never been given a value, the variable will be created and assigned the empty string, then the empty string will be returned.
- ◆ You can set the value of a variable by giving the value as the second argument, after the variable name. No value is returned by the “var” command in this case.
- ◆ You can assign values to multiple variables “in parallel” by giving successive name/value pairs.
- ◆ If invoked in a procedure on a variable that has not been declared global (using the “global” command), this applies to the local variable of the given name, even if it has no value yet.

See Also: global.

5.6 Coding

This section provides information about the features and commands of Tcl that are important to know when using Swat, and the features and commands of Swat that are important to know when using Tcl. These features should be kept in mind while programming in Tcl because, if used properly, they make programming, debugging, and understanding existing commands much easier. This section will contain the following parts:

- ◆ Swat Data Structures
Descriptions of the major data structures and the commands that access them.
- ◆ Examples



5.6.1 Swat Data Structure Commands

symbol, type, patient, handle, brk, cbrk, event, thread,
src, cache, table

This section contains information about Swat's built-in data structures and the commands that access them. These commands examine and modify vital information about the state of GEOS while it is running under Swat.

5.6

■ brk

Usage:

```
brk <addr> [<command>]
brk pset <addr> [<command>]
brk aset <addr> [<command>]
brk tset <addr> [<command>]
brk clear <break>*
brk delete <break>*
brk enable <break>*
brk disable <break>*
brk address <break>
brk list [<addr>]
brk debug [<flag>]
brk isset <addr>
brk cond <break> <condition>*
brk cmd <break> [<command>]
brk delcmd <break> [<command>]
```

Examples:

“brk WinOpen”

Sets the machine to stop unconditionally when any thread calls WinOpen.

“brk pset WinOpen”

Sets the machine to stop when any thread for the current patient calls WinOpen.

“brk tset WinOpen”

Sets the machine to stop when any thread for the current patient calls WinOpen, and deletes the breakpoint when the machine next stops.

Tool Command Language

304

“brk enable 1 3-5”

Re-enables breakpoints 1, 3, 4, and 5

“brk clear 2-”

Clears all breakpoints from number 2 onward.

“brk cond 3 cx=42”

Sets breakpoint 3 to be conditional, stopping when the machine reaches the breakpoint’s address with CX being 42.

5.6

“brk cond 2 (ss:0)!=1b80h”

Sets breakpoint 2 to be conditional, stopping when the machine reaches the breakpoint’s address with the word at ss:0 not being 1b80h. Note that the “ss” is the value of the **ss** register when the “brk cond” command is executed, not when the breakpoint is reached.

Synopsis:

Allows you to specify that execution should stop when it reaches a particular point. These breakpoints can be conditional, and can execute an arbitrary Tcl command, which can say whether the machine is to remain stopped, or continue on its way.

Notes:

- ◆ Once you’ve set a breakpoint, “brk” will return to you a token for that breakpoint that begins with “brk” and ends with a number. When you refer to the breakpoint, you can use either the full name (as you’ll usually do from a Tcl procedure), or just the number.
- ◆ Breakpoints have four attributes: the address at which they are set, the condition set on their being recognized, the Tcl command string to execute when they are recognized, and the Tcl command string to execute when they are deleted.
- ◆ The condition is set either when the breakpoint is set, using the “cbrk” command, or after you’ve set the breakpoint, by invoking the “brk cond” command.
- ◆ A breakpoint’s condition is evaluated (very quickly) on the PC and can check only word registers (the 8 general registers, the three segment registers other than CS, and the current thread; each register may be checked only once in a condition) and a single word of memory. Each `<condition>` argument is of the form “`<reg><op><value>`”. `<reg>` is one of the 16-bit machine registers, “thread” (for the current thread), or the address of a word of memory to check, enclosed in parentheses. `<op>` is a



relational operator taken from the following set:

= equal-to

!= not-equal-to

> < >= <= unsigned greater-than, less-than, greater-or-equal, and less-or-equal

+> +< +>= +<= signed greater-than, less-than, greater-or-equal, and less-or-equal

<value> is a regular Swat address expression. If it is handle-relative, and the <reg> is one of the three non-CS segment registers, the condition will be for the segment of that handle and will change automatically as the handle's memory shifts about on the heap. Similar things will happen if you specify a number as the <value> for a segment register and the number is the current segment of a block on the heap.

5.6

- ◆ If you give no <condition> argument to the “brk cond” command, you will remove any condition the breakpoint might have, making it, therefore, unconditional.
- ◆ If a breakpoint is given an associated <command> string, it will be evaluated before the breakpoint is taken. If the result of the evaluation is an error, a non-numeric string, or a numeric string that's non-zero, the breakpoint will be taken. Otherwise, the machine will be allowed to continue (so long as no other breakpoint command or other part of Swat insists that it remain stopped). You can use this to simply print out information when execution reaches the breakpoint address without interrupting the machine's execution.
- ◆ The global variable “breakpoint” contains the name of the breakpoint whose command is being evaluated while that command is being evaluated.
- ◆ You can change the command associated with a breakpoint with the “brk cmd” command. If you give no <command> argument, then no command will be executed and the breakpoint will always be taken, so long as any associated condition is also met.
- ◆ If a breakpoint has both a condition and a command, the command will not be executed until the condition has been met, unless there's another breakpoint at the same address with a different, or no, condition.
- ◆ You can set a breakpoint to last only during the next continuation of the machine by calling “brk tset”. The breakpoint thus set will be removed when next the machine comes to a full stop, regardless of why it stopped

(i.e. if it hits a different breakpoint, the temporary breakpoint will still be removed). The breakpoint will only be taken if the thread executing when it is hit is owned by the patient that was current when the breakpoint was set.

5.6

- ◆ Each <break> argument to the “brk clear”, “brk enable” and “brk disable” commands can be either a single breakpoint token (or number), or a range of the form <start>-<end>, where either <start> or <end> may be absent. If <start> is missing, the command affects all breakpoints from number 1 to <end>. If <end> is missing, the command affects all breakpoints from <start> to the last one in existence.
- ◆ If you give no <break> argument to “brk clear”, “brk enable” or “brk disable”, the command will apply to all breakpoints that are specific to the current patient, i.e. that were set with the “brk pset” command, unless the current patient is the kernel, in which case they will apply to all breakpoints that are specific to no patient (i.e. those set with the “brk aset” or “brk <addr>” commands).
- ◆ “brk address” returns the address expression for where the breakpoint is set. This will usually be of the form ^h<handle-id>:<offset>, with both <handle-id> and <offset> in hex (followed by an “h”, of course). If the breakpoint is set at an absolute address, you will get back only a single hex number, being the linear address at which the breakpoint is set.
- ◆ If you type “brk list” with no argument, Swat will print out a listing of the currently-active breakpoints. If you give an <addr> (address expression) argument, however, you’ll be returned a list of the breakpoints set at the given address. If there are no breakpoints there, the list will be empty.
- ◆ As a shortcut, you can invoke “brk isset” to see if any breakpoints are set at the given address, if you’re not interested in which ones they are.

■ cache

Usage:

```
cache create (lru | fifo) <maxSize> [<flushProc>]
cache destroy <cache> [flush | noflush]
cache lookup <cache> <key>
cache enter <cache> <key>
cache invalone <cache> <entry>
cache invalall <cache> [flush | noflush]
cache key <cache> <entry>
cache size <cache>
cache maxsize <cache>
```



```
cache setmaxsize <cache> <maxSize>
cache getval <cache> <entry>
cache setval <cache> <entry> <value>
```

Examples:

```
"var cache [cache create lru 10]"
```

Creates a cache of 10 items that are flushed on a least-recently-used basis. The returned token is saved for later use.

5.6

```
"var entry [cache lookup $cache mom]"
```

Sees if an entry with the key "mom" is in the cache and saves its entry token if so.

```
"echo mom=[cache getval $cache $entry]"
```

Retrieves the value stored in the entry for "mom" and echoes it.

```
"cache invalone $cache $entry"
```

Flushes the entry just found from the cache.

```
"cache destroy $cache"
```

Destroys the cache.

Synopsis:

The cache command, as the name implies, maintains a cache of data that is keyed by strings. When a new entry is added to an already-full cache, an existing entry is automatically flushed based on the usage message with which the cache was created: *lru* (last recently used) or *fifo* (first in, first out). If *lru*, the least-recently-used entry is flushed; if *fifo*, the oldest entry is flushed.

Notes:

- ◆ Unlike the "table" command, the "cache" command returns tokens for entries, not their values. This allows entries to be individually flushed or their values altered.
- ◆ If a <flushProc> is specified when the cache is created, the procedure will be called each time an entry is flushed from the cache. It will be called "<flushProc> <cache> <entry>" where <cache> is the token for the cache, and <entry> is the token for the entry being flushed.
- ◆ If the maximum size of a full cache is reduced, entries will be flushed from the cache to bring it down to the new maximum size. The <flushProc> will be called for each of them.

5.6

- ◆ If the values stored in the cache entries should not be freed when the cache is destroyed, pass “noflush” to “cache destroy”. The default is to flush (and hence call the <flushProc>) all entries from the cache before it is destroyed.
- ◆ If the values stored in the cache entries should not be freed when the cache is flushed, pass “noflush” to “cache invalall”. The default is to call the <flushProc> for each entry in the cache before it is actually flushed.
- ◆ If an entry is not found in the cache, “cache lookup” will return an empty string.
- ◆ When an entry is created, “cache enter” returns a 2-list containing the entry token as its first element, and an integer, as its second element, that is either non-zero or 0, to tell if the entry is new or was already present, respectively.

■ cbrk

Usage:

```
cbrk <addr> <condition>*  
cbrk aset <addr> <condition>*  
cbrk tset <addr> <condition>*  
cbrk clear <break>*  
cbrk delete <break>*  
cbrk enable <break>*  
cbrk disable <break>*  
cbrk address <break>  
cbrk list [<addr>]  
cbrk debug [<flag>]  
cbrk isset <addr>  
cbrk cond <break> <condition>*  
cbrk cmd <break> [<command>]  
cbrk delcmd <break> [<command>]
```

Examples:

```
“cbrk WinOpen di=1b80h”
```

Stops the machine when execution reaches WinOpen() with **di** set to 1b80h.

Synopsis:

Allows you to set fast conditional breakpoints.

Notes:

- ◆ All these subcommands function the same as for the “brk” command, with the exception of the “aset” and “tset” commands, which expect the condition for the breakpoint, rather than an associated command.
- ◆ There are a limited number of these sorts of breakpoints that can be set in the PC (currently 8), so they should be used mostly for heavily-travelled areas of code (e.g. inner loops, or functions like **ObjCallMethodTable()** in the kernel).
- ◆ For more information on the subcommands and the format of arguments, see the documentation for the “brk” command.

5.6

■ event

event <subcommand>

The **event** command provides access to Swat’s internal events. The subcommands are as follows:

handle <eventName> <handler> [<data>]

The <handler> procedure is invoked each time an event of type <eventName> is dispatched. The handler receives two arguments: an event-specific piece of data, and the given <data>. A handler procedure should be declared

```
proc <handler> {arg data} {<body>}
```

The **handle** subcommand returns an <event> for later use in deleting it. The <handler> should return one of **event_handled**, **event_not_handled**, or **event_stop_handling**. If it returns **event_stop_handling**, the event will not be dispatched to any other handlers of the event.

delete <event>

Deletes the given event handler given by the **event handle** command.

dispatch <eventName> <arg>

Dispatches the given event with the given <arg> to all handlers of that event. If <eventName> is a pre-defined event type, <arg> will be converted to the appropriate type before being dispatched. Otherwise it is passed as a string.

create Returns a number that represents a new event type. Handlers may then be defined for and events dispatched of the new type.

list Lists all Tcl-registered events by event-name and handler function.

The events which are currently defined are:

FULLSTOP

Generated when patient stops for a while. Argument is string telling why the patient stopped.

CONTINUE

Generated just before the patient is continued. The argument is non-zero if going to single-step.

5.6

TRACE

Generated when the execution of a source line completes and the patient is in line-trace mode.

START

Generated when a new patient/thread is created. Argument is patient token of the patient involved.

STACK

Current stack frame has changed. The argument is non-zero if the stack change comes from a change in patients/threads or zero if the change comes from actually going up or down the stack in the current patient.

DETACH

Detaching from the PC. The argument is always zero.

RESET

Returning to the top level. The argument is always zero.

ATTACH

Attached to the PC. The argument is always zero.

RELOAD

Kernel was reloaded. The argument is always zero.

CHANGE

Current patient has changed. The argument is the token for the previous patient.

STEP

Machine has stepped a single instruction. The argument is the value to pass to **patient stop** if you wish the machine to stay stopped.

STOP

Machine has hit a breakpoint. The argument is the value to pass to **patient stop** if you wish the machine to stay stopped.

INT

Machine has hit some other interrupt that's being caught. The argument is the interrupt number. The machine will remain stopped unless it is continued with `continue-patient`.

■ handle

Usage: handle lookup <id>
 handle find <address>
 handle all
 handle nointerest <interest-record>



```
handle interest <handle> <proc> [<data>+]  
handle segment <handle>  
handle size <handle>  
handle state <handle>  
handle owner <handle>  
handle patient <handle>  
handle other <handle>  
handle id <handle>  
handle isthread <handle>  
handle iskernel <handle>  
handle isfile <handle>  
handle isvm <handle>  
handle ismem <handle>
```

5.6

Examples:

```
“handle lookup [read-reg bx]”  
    get the handle token for the handle whose ID is in the BX  
    register.  
“handle interest $h ob-interest-proc [concat si=$chunk $message]”  
    call ob-interest-proc, passing the list {si=$chunk $message},  
    whenever the state of the handle whose token is in $h changes.  
“handle patient $h”  
    get the token for the patient that owns the handle whose token  
    is in $h  
“handle all”  
    get the list of the ID’s of all handles currently in Swat’s handle  
    table.
```

Synopsis:

The “handle” command provides access to the structures Swat uses to track memory and thread allocation on the PC.

Notes:

- ◆ As with most other commands that deal with Swat structures, you use this one by calling a lookup function (the “lookup” and “find” subcommands) to obtain a token that you use for further manipulations. A handle token is also returned by a few other commands, such as `addr-parse`.

5.6

- ◆ Handle tokens are valid only until the machine is continued. If you need to keep the token for a while, you will need to register interest in the handle using the “interest” subcommand. Most handles tokens will simply be cached while the machine is stopped and flushed from the cache when the machine continues. Only those handles for which all state changes must be known remain in Swat’s handle table. For example, when a conditional breakpoint has been registered with the stub using the segment of a handle, the condition for that breakpoint must be updated immediately should the memory referred to by the handle be moved, swapped or discarded. Keeping the number of tracked handles low reduces the number of calls the stub must make to tell Swat about handle-state changes.
- ◆ The <id> passed to the “lookup” subcommand is an integer. Its default radix is decimal, but you can specify the radix to use in all the usual ways. The value returned is the token to use to obtain further information about the handle.
- ◆ “handle size” returns the number of bytes allocated to the handle.
- ◆ “handle segment” returns the handle’s segment (if it’s resident) in decimal, as it’s intended for use by Tcl programs, not people.
- ◆ “handle owner” returns the token of the handle that owns the given handle, not its ID.
- ◆ “handle all” returns a list of handle ID numbers not a list of handle tokens. The list is only those handles currently known to Swat.
- ◆ “handle interest” tells Swat you wish to be informed when the handle you pass changes state in some way. The procedure <proc> will be called with two or more arguments. The first is the token of the handle whose state has changed, and the second is the state change the handle has undergone, taken from the following set of strings:

swapin	Block swapped in from disk/memory
load	Resource freshly loaded from disk
swapout	Block swapped to disk/memory
discard	Block discarded
resize	Block changed size and maybe moved
move	Block moved on heap



free Block has been freed

fchange Block's **HeapFlags** changed

Any further arguments are taken from the <data>+ arguments provided when you expressed interest in the handle. This subcommand returns a token for an interest record that you pass to “handle nointerest” when you no longer care about the handle. When the block is freed (the state change is “free”), there is no need to call “handle nointerest” as the interest record is automatically deleted.

5.6

- ◆ “handle state” returns an integer indicating the state of the handle. The integer is a mask of bits that mean different things:

Table 5-3 *The State Subcommand: Block Information*

Mask	State	Mask	State
0xf8000	Type	0x00200	Attached
0x00040	Discarded	0x00008	Fixed
0x00001	Resident	0x00800	LMem
0x00100	Process	0x00020	Swapped
0x00004	Discardable	0x00400	Kernel
0x00080	Resource	0x00010	Shared
0x00002	Swapable		

When the integer is AND-ed with the mask for Type (0xf8000), the following values indicate the following types of handles:

Table 5-4 *The State Subcommand: Block Type*

Mask	State
0xe0000	Thread
0xb0000	Semaphore
0x80000	Event with stack data chain
0x70000	Stack data chain element
0xd0000	File
0xa0000	Saved block
0x08000	Memory
0xc0000	VM File
0x90000	Event
0x60000	Timer
0x40000	Event queue

- ◆ “handle other” returns the handle’s otherInfo field. Note: This isn’t necessarily the otherInfo field from the PC. E.g., for resource handles, it’s the symbol token of the module for the handle.

■ patient

Usage:

patient find <name>
patient name [<patient>]
patient fullname [<patient>]
patient data [<patient>]
patient threads [<patient>]
patient resources [<patient>]
patient libs [<patient>]
patient path [<patient>]
patient all
patient stop [<addr>]

5.6

Examples:

“patient find geos”

Returns the patient token for the kernel, if it’s been loaded yet.

“patient fullname \$p”

Returns the permanent name for the patient whose token is stored in the variable p.

“patient stop \$data”

Tells the dispatcher of the STEP event that it should keep the machine stopped when the STEP event has been handled by everyone.

Synopsis: This command provides access to the various pieces of information that are maintained for each patient (geode) loaded by GEOS.

Notes:

- ◆ Subcommands may be abbreviated uniquely.
- ◆ Swat always has the notion of a “current patient”, whose name is displayed in the prompt. It is this patient that is used if you do not provide a token to one of the subcommands that accepts a patient token.
- ◆ “patient name” returns the name of a patient. The name is the non-extension portion of the geode’s permanent name. It will have a number added to it if more than one instance of the geode is active on the PC.



Thus, if two GeoWrites are active, there will be two patients in Swat: “write” and “write2”.

- ◆ “patient fullname” returns the full permanent name of the patient. It is padded with spaces to make up a full 12-character string. This doesn’t mean you can obtain the non-extension part by extracting the 0th element of the result with the “index” command, however; you’ll have to use the “range” command to get the first 8 characters, then use “index” to trim the trailing spaces off, if you want to.
- ◆ “patient data” returns a three-element list: {<name> <fullname> <thread-number>} <name> and <fullname> are the same as returned by the “name” and “fullname” subcommands. <thread-number> is the number of the current thread for the patient. Each patient has a single thread that is the one the user looked at most recently, and that is its current thread. The current thread of the current patient is, of course, the current thread for the whole debugger.
- ◆ “patient threads” returns a list of tokens, one for each of the patient’s threads, whose elements can be passed to the “thread” command to obtain more information about the patient’s threads (such as their numbers, handle IDs, and the contents of their registers).
- ◆ “patient resources” returns a list of tokens, one for each of the patient’s resources, whose elements can be passed to the “handle” command to obtain more information about the patient’s resources (for example, their names and handle IDs).
- ◆ “patient libs” returns a list of patient tokens, one for each of the patient’s imported libraries. The kernel has all the loaded device drivers as its “imported” libraries.
- ◆ “patient path” returns the absolute path of the patient’s executable.
- ◆ “patient all” returns a list of the tokens of all the patients known to Swat.
- ◆ “patient stop” is used only in STEP, STOP and START event handlers to indicate you want the machine to remain stopped once the event has been dispatched to all interested parties. <addr> is the argument passed in the STEP and STOP events. A START event handler should pass nothing.
- ◆ A number of other commands provide patient tokens. “patient find” isn’t the only way to get one.

5.6

■ src

Usage:

```
src line <addr>
src read <file> <line>
src cache [<max>]
src addr <file> <line> [<patient>]
```

Examples:

5.6

“src line cs:ip”

Returns a two-list holding the source-line number, and the absolute path of the file in which it lies (not in this order), that encompasses CS:IP.

“src read /pcgeos/appl/sdk_c/hello/hello.goc 64”

Reads the single given source line from the given file.

“src addr icdecode.c 279”

Returns an address-list for the start of the code produced for the given line.

“src cache 10”

Allow 10 source files to be open at a time. This is the default.

Synopsis:

The “src” command allows the Tcl programmer to manipulate the source- line maps contained in all the geodes’ symbol files.

Notes:

- ◆ The “src line” commands returns its list as {<file> <line>}, with the <file> being absolute. If no source line can be found, the empty list is returned.
- ◆ The <file> given to the “src read” command must be absolute, as the procedure using this command may well be wrong as to Swat’s current directory. Typically this name will come from the return value of a “src line” command, so you needn’t worry.
- ◆ The line returned by “src read” contains no tabs and does not include the line terminator for the line (the <lf> for UNIX, or the <cr><lf> pair for MS-DOS).
- ◆ “src addr” returns an address-list, as returned from “addr-parse”, not an address expression, as you would *pass* to “addr-parse”. If the <file> and <line> cannot be mapped to an address, the result will be the empty list.



- ◆ The <file> given to “src addr” must be the name that was given to the assembler/compiler. This includes any leading path if the file wasn’t in the current directory when the assembler/compiler was run.
- ◆ “src cache” returns the current (or new) number of open files that are cached.

■ symbol

Usage:

```
symbol find <class> <name> [<scope>]
symbol faddr <class> <addr>
symbol match <class> <pattern>
symbol scope <symbol>
symbol name <symbol>
symbol fullname <symbol>
symbol class <symbol>
symbol type <symbol>
symbol get <symbol>
symbol patient <symbol>
symbol tget <symbol>
symbol addr <symbol>
symbol foreach <scope> <class> <callback> [<data>]
```

5.6

Examples:

```
“symbol find type LMemType”
    Locate a type definition named LMemType

“symbol faddr proc cs:ip”
    Locate the procedure in which CS:IP lies.

“symbol faddr {proc label} cs:ip”
    Locate the procedure or label just before cs:ip.

“symbol fullname $sym”
    Fetch the full name of the symbol whose token is in the $sym
    variable.

“symbol scope $sym”
    Fetch the token of the scope containing the passed symbol. This
    will give the structure containing a structure field, or the
    procedure containing a local variable, for example.
```

Synopsis: Provides information on the symbols for all currently-loaded patients. Like many of Swat's commands, this operates by using a lookup function (the "find", "faddr", "match", or "foreach" subcommands) to obtain a token for a piece of data that's internal to Swat. Given this token, you then use the other subcommands (such as "name" or "get") to obtain information about the symbol you looked up.

Notes:

5.6

- ◆ There are many types of symbols that have been grouped into classes that may be manipulated with this command. For a list of the symbol types and their meaning, type "help symbol-types". The type of a symbol can be obtained with the "symbol type" command.

- ◆ The symbol classes are as follows:

type	describes any structured type: typedef, struct, record, etype, union. Symbols of this class may also be used in place of type tokens (see the "type" command).
field	describes a field in a structured type: field, bitfield.
enum	describes a member of an enumerated type: enum, message.
const	a constant defined with EQU: <i>const</i> .
var	describes any variable symbol: var, chunk, locvar, class, masterclass, variantclass.
locvar	describes any local variable symbol: locvar, locstatic.
scope	describes any symbol that holds other symbols within it: module, proc, blockstart, struct, union, record, etype.
proc	describes only proc symbols.
label	describes any code-related symbol: label, proc, loclabel.
onstack	describes only symbols created by the directive.
module	describes only segment/group symbols.
profile	describes a symbol that marks where profiling code was inserted by a compiler or assembler.

- ◆ The <class> argument for the "find", "faddr" and "match" subcommands may be a single class, or a space-separated list of classes. For example, "symbol faddr {proc label} CS:IP" would find the symbol closest to CS:IP



(but whose address is still below or equal to CS:IP) that is either a procedure or a label.

- ◆ The “symbol find” command locates a symbol given its name (which may be a symbol path).
- ◆ The “symbol faddr” command locates a symbol that is closest to the passed address.
- ◆ A symbol’s “fullname” is the symbol path, from the current patient, that uniquely identifies the symbol. Thus if a procedure-local variable belongs to the current patient, the fullname would be
`<segment>::
 where <segment> is the segment holding the <procedure>, which is the procedure for which the local variable named <name> is defined.`
- ◆ You can force the prepending of the owning patient to the fullname by passing `<with-patient>` as a non-empty argument (“yes” or “1” are both fine arguments, as is “with-patient”).
- ◆ The “symbol get” commands provides different data for each symbol class, as follows:

5.6

var, locvar, chunk: {<addr> <class> <type>}

`<addr>` is the symbol’s address as for the “addr” subcommand, `<class>` is the storage class of the variable and is one of static (a statically allocated variable), lmem (an lmem chunk), local (a local variable below the frame pointer), param (a local variable above the frame pointer), or reg (a register variable; address is the machine register number -- and index into the list returned by the “current-registers” command).

object class: {<addr> <class> <type> <flag> <super>}

first three elements same as for other variables. `<flag>` is “variant” if the class is a variant class, “master” if the class is a master class, or empty if the class is nothing special. `<super>` is the symbol token of the class’s superclass.

label-class: {<addr> (near | far)}

`<addr>` is the symbol’s address as for the “addr” subcommand. The second element is “near” or “far” depending on the type of label involved.

field-class: {<bit-offset> <bit-width> <field-type> <struct-type>}

Tool Command Language

320

5.6

<bit-offset> is the offset of the field from the structure/union/record's base expressed in bits. <bit-width> is the width of the field, in bits. <field-type> is the type for the field itself, while <struct-type> is the token for the containing structured type.

const: {<value>}
<value> is just the symbol's value.

enum-class: {<value> <etype>}
<value> is the symbol's value. <etype> is the enumerated type's symbol.

blockstart, blockend: {<addr>}
<addr> is the address bound to the symbol.

onstack: {<addr> <data>}
<addr> is the address at which the ON_STACK was declared.
<data> is the arguments given to the ON_STACK directive.

module: {<patient>}
<patient> is the token for the patient owning the module.

- ◆ A related command, “symbol tget” will fetch the type token for symbols that have data types (var-, field- and enum-class symbols).
- ◆ “symbol addr” can be used to obtain the address of symbols that actually have one (var-, locvar- and label-class symbols). For locvar symbols, the address is an offset from the frame pointer (positive or negative). For var- and label-class symbols (remember that a procedure is a label-class symbols), the returned integer is the offset of the symbol within its segment.
- ◆ “symbol patient” returns the token of the patient to which the symbol belongs.
- ◆ “symbol foreach” will call the <callback> procedure for each symbol in <scope> (a symbol token) that is in one of the classes given in the list <class>. The first argument will be the symbol token itself, while the second argument will be <data>, if given. If <data> wasn't provided, <callback> will receive only 1 argument. <callback> should return 0 to continue iterating, or non-zero to stop. A non-integer return is assumed to mean stop. “symbol foreach” returns whatever the last call to <callback> returned.
- ◆ By default, “symbol scope” will return the physical scope of the symbol. The physical scope of a symbol is the symbol for the segment in which the



symbol lies, in contrast to the lexical scope of a symbol, which is where the name of the symbol lies. The two scopes correspond for all symbols but static variables local to a procedure. To obtain the lexical scope of a symbol, pass <lexical> as a non-zero number.

See Also: symbol-types, type

■ table

Usage: table create [<initBuckets>]
 table destroy <table>
 table enter <table> <key> <value>
 table lookup <table> <key>
 table remove <table> <key>

5.6

Examples:

“var kitchen [table create 32]”

Create a new table with 32 hash buckets initially.

“table enter \$t tbrk3 {1 2 3}”

Enter the value “1 2 3” under the key “tbrk3” in the table whose token is stored in the variable t.

“table lookup \$t tbrk4”

Fetch the value, if any, stored under the key “tbrk4” in the table whose token is stored in the variable t.

“table remove \$t tbrk3”

Remove the data stored in the table, whose token is stored in the variable t, under the key “tbrk3”

“table destroy \$t”

Destroy the table \$t and all the data stored in it.

Swat Display 5-1 The table Structure

```
(mess1:0) 159 => var yearTable [table create]
(mess1:0) 160 => table enter $yearTable synclavier 1979
(mess1:0) 161 => table enter $yearTable moog 1966
(mess1:0) 162 => table lookup $yearTable synclavier
1979
(mess1:0) 163 => var yearTable
1403188
(mess1:0) 164 => table lookup 1403188 moog
```



Tool Command Language

322

```
1966
(mess1:0) 165 => table remove $yearTable synclavier
(mess1:0) 166 => table lookup $yearTable synclavier
nil
(mess1:0) 167 => table destroy $yearTable
```

5.6

Synopsis: The “table” command is used to create, manipulate and destroy hash tables. The entries in the table are keyed on strings and contain strings, as you’d expect from Tcl.

Notes:

- ◆ The <initBuckets> parameter to “table create” is set based on the number of keys you expect the table to have at any given time. The number of buckets will automatically increase to maintain hashing efficiency, should the need arise, so <initBuckets> isn’t a number that need be carefully chosen. It’s best to start with the default (16) or perhaps a slightly larger number.
- ◆ If no data are stored in the table under <key>, “table lookup” will return the string “nil”, for which you can test with the “null” command.

■ thread

Usage: thread id <thread>
thread register <thread> <regName>
thread handle <thread>
thread endstack <thread>
thread number <thread>
thread all

Examples:

“thread register \$t cx”
Fetches the value for the CX register for the given thread.

“thread number \$t”
Fetches number swat assigned to thread when it was first encountered.



Swat Display 5-2 The thread Structure

```
(mess1:0) 145 => patient threads
2667104
(mess1:0) 146 => thread id 2667104
11184
(mess1:0) 147 => thread all
767532 756068 1348520 1348868 1349216 1349748 1350236 1402096 1079392 2667104
(mess1:0) 148 => thread handle 756068
880428
(mess1:0) 149 => thread number 756068
0
```

5.6

Synopsis: Returns information about a thread, given its thread token. Thread tokens can be obtained via the “patient threads” command, or the “handle other” command applied to a thread handle’s token.

Notes:

- ◆ Subcommands may be abbreviated uniquely.
- ◆ “thread id” returns the handle ID, in decimal, of the thread’s handle. This is simply a convenience.
- ◆ “thread register” returns the contents of the given register in the thread when it was suspended. All registers except “pc” are returned as a single decimal number. “pc” is returned as two hexadecimal numbers separated by a colon, being the cs:ip for the thread. Note that GEOS doesn’t actually save the AX and BX registers when it suspends a thread, at least not where Swat can consistently locate them. These registers will always hold 0xadeb unless the thread is the current thread for the machine (as opposed to the current thread for swat).
- ◆ “thread handle” returns the token for the thread’s handle.
- ◆ “thread endstack” returns the maximum value SP can hold for the thread, when it is operating off its own stack. Swat maintains this value so it knows when to give up trying to decode the stack.
- ◆ “thread number” returns the decimal number Swat assigned the thread when it first encountered it. The first thread for each patient is given the number 0 with successive threads being given the highest thread number known for the patient plus one.

- ◆ “thread all” returns a list of tokens for all the threads known to Swat (for all patients).

■ type

Usage:

5.6

```
type <basic-type-name>
type make array <length> <base-type>
type make pstruct (<field> <type>)+
type make struct (<field> <type> <bit-offset> <bit-length>)+
type make union (<field> <type>)+
type make <ptr-type> <base-type>
type delete <type>
type size <type>
type class <type>
type name <type> <var-name> <expand>
type aget <array-type>
type fields <struct-type>
type members <enum-type>
type pget <ptr-type>
type emap <num> <enum-type>
type signed <type>
type field <struct-type> <offset>
type bfget <bitfield-type>
```

Examples:

“type word” Returns a type token for a word (2-byte unsigned quantity).

“type make array 10 [type char]”

Returns a type token for a 10-character array.

“type make optr [symbol find type GenBase]”

Returns a type token for an optr (4-byte global/local handle pair) to a “GenBase” structure.

Synopsis:

Provides access to the type descriptions by which all PC-based data are manipulated in Swat, and allows a Tcl procedure to obtain information about a type for display to the user, or for its own purposes. As with other Swat commands, this works by calling one subcommand to obtain an opaque “type token”, which you then pass to other commands.

Notes:

- ◆ Type tokens and symbol tokens for type-class symbols may be freely interchanged anywhere in Swat.
- ◆ There are 11 predefined basic types that can be given as the `<basic-type-name>` argument in “type `<basic-type-name>`”. They are: **byte** (single-byte unsigned integer), **char** (single-byte character), **double** (eight-byte floating-point), **dword** (four-byte unsigned integer), **float** (four-byte floating-point), **int** (two-byte signed integer), **long** (four-byte signed integer), **sbyte** (single-byte signed integer), **short** (two-byte signed integer), **void** (nothing, useful as the base type for a pointer type), and **word** (two-byte unsigned integer) 5.6
- ◆ Most type tokens are obtained, via the “symbol get” and “symbol tget” commands, from symbols that are defined for a loaded patient. These are known as “external” type descriptions. “Internal” type descriptions are created with the “type make” command and should be deleted, with “type delete” when they are no longer needed.
- ◆ An internal structure type description can be created using either the “pstruct” (packed structure) or “struct” subcommands. Using “pstruct” is simpler, but you have no say in where each field is placed (they are placed at sequential offsets with no padding between fields), and all fields must be a multiple of 8 bits long. The “struct” subcommand is more complex, but does allow you to specify bitfields.
- ◆ “type make pstruct” takes 1 or more pairs of arguments of the form “`<field> <type>`”, where `<field>` is the name for the field and `<type>` is a type token giving the data type for the field. All fields must be specified for the structure in this call; fields cannot be appended to an existing type description.
- ◆ “type make struct” takes 1 or more 4-tuples of arguments of the form “`<field> <type> <bit-offset> <bit-length>`”. `<field>` is the name of the field, and `<type>` is its data type. `<bit-offset>` is the offset, in bits, from the start of the structure (starting with 0, as you’d expect). `<bit-length>` is the length of the field, in bits (starting with 1, as you’d expect). For a bitfield, `<type>` should be the field within which the bitfield is defined. For example, the C declaration:

```
struct {
    word a:6;
    word b:10;
    word c;
}
```

would result in the command “type make struct a [type word] 0 6 b [type

word] 6 10 c [type word] 16 16”, because a and b are defined within a word type, and c is itself a word.

- ◆ “type make union” is similar to “type make pstruct”, except all fields start at offset 0. Like “pstruct”, this cannot be used to hold bitfields, except by specifying a type created via “type make struct” command as the <type> for one of the fields.
- ◆ “type make array <length> <base-type>” returns a token for an array of <length> elements of the given <base-type>, which may be any valid type token, including another array type.
- ◆ “type make <ptr-type> <base-type>” returns a token for a pointer to the given <base-type>. There are 6 different classes of pointers in GEOS:

- nptr** a near pointer. 16-bits. points to something in the same segment as the pointer itself.
- fpnr** a far pointer. 32-bits. segment in high word, offset in the low.
- sptr** a segment pointer. 16-bits. contains a segment only.
- lpnr** an lmem pointer. 16-bits. contains a local-memory “chunk handle”. data pointed to is assumed to be in the same segment as the lpnr itself, but requires two indirections to get to it.
- hpnr** a handle pointer. 16-bits. a GEOS handle.
- opnr** an object pointer. 32-bits. contains a GEOS memory handle in the high word, and a GEOS local-memory chunk handle in the low.

- ◆ “type delete” is used to delete a type description created by “type make”. You should do this whenever possible to avoid wasting memory.
- ◆ Any type created by the “type make” command is subject to garbage collection unless it is registered with the garbage collector. If you need to keep a type description beyond the end of the command being executed, you must register it. See the “gc” command for details.
- ◆ “type size” returns the size of the passed type, in bytes.
- ◆ “type class” returns the class of a type, a string in the following set:

- char** for the basic “char” type only.
- int** any integer, signed or unsigned.



struct	a structure, record, or union.
enum	an enumerated type.
array	an array, of course,
pointer	a pointer to another type.
void	nothingness. Often a base for a pointer.
function	a function, used solely as a base for a pointer.
float	a floating-point number.

5.6

- ◆ Each type class has certain data associated with it that can only be obtained by using the proper subcommand.
- ◆ “type aget” applies only to an array-class type token. It returns a four-element list: {<base-type> <low> <high> <index-type>} <base-type> is the type token describing elements of the array. <low> is the lower bound for an index into the array (currently always 0), <high> is the inclusive upper bound for an index into the array, and <index-type> is a token for the data type that indexes the array (currently always [type int]).
- ◆ “type fields” applies only to a struct-class type token. It returns a list of four-tuples {<name> <offset> <length> <type>}, one for each field in the structure. <offset> is the *bit* offset from the start of the structure, while <length> is the length of the field, again in *bits*. <type> is the token for the data type of the field, and <name> is, of course, the field’s name.
- ◆ “type members” applies only to an enum-class type token. It returns a list of {<name> <value>} pairs for the members of the enumerated type.
- ◆ “type pget” applies only to a pointer-class type token. It returns the type of pointer (“near”, “far”, “seg”, “lmem”, “handle”, or “object”) and the token for the type to which it points.
- ◆ “type bfget” returns a three-list for the given bitfield type: {<offset> <width> <is-signed>}
- ◆ “type signed” returns non-zero if the type is signed. If the <type> is not an int-class type, it is considered unsigned.
- ◆ “type emap” can be used to map an integer to its corresponding enumerated constant. If no member of the enumerated type described by <type> has the value indicated, “nil” is returned, else the name of the matching constant is returned.

5.6

- ◆ “type field” maps an offset into the passed struct-class type into a triple of the form {<name> <length> <ftype>}, where <name> can be either a straight field name, or a string of the form <field>.<field>... with as many .<field> clauses as necessary to get to the smallest field in the nested structure <type> that covers the given byte <offset> bytes from the start of the structure. <length> is the *bit* length of the field, and <ftype> is its type.
- ◆ “type name” produces a printable description of the given type, using C syntax. <varname> is the name of the variable to which the type belongs. It will be placed at the proper point in the resulting string. If <expand> is non-zero, structured types (including enumerated types) are expanded to display their fields (or members, as the case may be).

See Also: gc, symbol, symbol-types, value

5.6.2 Examples

This section will contain a few examples of Tcl code for Swat commands, showing the use of some of included Tcl commands. A good way to view the code for a particular procedure is to type:

```
info body <procname>
```

on the Swat command line. This will print out the body of the given <procname> . One thing to watch out for, however, is the case when a procedure has not been loaded into Swat yet (i.e. it has not been used yet). If this is the case, Swat will have no information about the procedure and will thus print nothing. The command must be loaded into Swat either with the **load** command, or by just typing the command name which will usually autoload the command. (See section 5.7 on page 330.) Then the **info body** <procname> command can be used.

Some code examples:

Swat Display 5-3 The Whatat Command

```
[defcommand whatat {addr} output
{Given an address, print the name of the variable at that address}
{
var a [sym faddr var $addr]
```



```

    if {[null $a]}{
        echo *nil*
    } else {
        echo [sym name $a]
    }
}]]

```

*This example shows the code of the **whatat** command. Note the use of the **sym** (an abbreviation for **symbol**) command to find the address of the given variable **<addr>** of class **<var>**.*

5.6

Swat Display 5-4 The Bytes Command

```

1      var addr [get-address $addr ds:si]
2      var base [index [addr-parse $addr] 1]
3      echo {Addr: +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +a +b +c +d +e +f}
4      #fetch the bytes themselves
5      var bytes [value fetch $addr [type make array $num [type byte]]]
6      #
      # $s is the index of the first byte to display on this row, $e is the
      # index of the last one. $e can get > $num. The loop handles this case.
      #
      var s 0 e [expr 16-($base&0xf)-1]
      #
      # $pre can only be non-zero for the first line, so set it once here.
      # We'll set it to zero when done with the first line.
      # $post can be non-zero only for the last line, but we can't just
      # set it to zero and let the loop handle it, as the first may be the
      # last, so...
      #
      var pre [expr 16-($e-$s)-1]
      if {$e > $num} {
          var post [expr $e-($num-1)]
      } else {
          var post 0
      }

      [for {var start [expr {$base&~0xf}]}
        {$s < $num}
        {var start [expr $start+16]}
        {
28      #extract the bytes we want
29      var bs [range $bytes $s $e]

```

```
30      echo [format {%04xh: %*s%s%*s "%*s%s%*s"} $start
[expr $pre*3] {}
[map i $bs {format %02x $i}]
[expr $post*3] {}
$pre {}
[mapconcat i $bs {
if {$i >= 32 && $i < 127} {
format %c $i
} else {
format .
}
}]
$post {}]
var s [expr $e+1] e [expr $e+16] pre 0
if {$e >= $num} {
var post [expr $e-($num-1)]
}
}]
set-address $addr+$num-1
set-repeat [format {%0 {s} $2} $addr+$num]
```

*This example shows the code for the **bytes** commands. Notice the use of the **type** command on the fifth line, and the **range** command on the twenty-ninth line.*

5.7 Using a New Command

Once a new command is written, it needs to be loaded into Swat so that it can be used. Depending on how the command is to be used, you may be interested in any of the following topics:

- ◆ Compilation
- ◆ Autoloading
- ◆ Explicit loading



5.7.1 Compilation

It is possible to byte-compile a Tcl script. The **bc** Tcl command creates a .TLC file containing compiled Tcl code—this code will run faster than normal Tcl code. When loading, Swat will load a .TLC file instead of a .TCL file where possible. Making changes to compiled Tcl functions involves changing the source code and re-compiling.

5.7

5.7.2 Autoloading

If the development environment has been set up properly, there should already exist the **/pcgeos/Tools/swat/lib** directory on the workstation. This directory will contain all of the code files for the built-in Swat commands. To autoload a new command, copy its code file to the **/pcgeos/Tools/swat/lib** directory and add its name to the **autoload.tcl** file in the directory. This will load the command into Swat every time Swat is started. For example, say the command **blitzburp** has just been written to examine a new data structure. First, copy the file containing its code (say **blitz.tcl**) into the **/pcgeos/Tools/swat/lib** directory. Next, edit the **autoload.tcl** file and add *one* of the following lines:

```
[autoload blitzburp 0 blitz]  
[autoload blitzburp 1 blitz]
```

This will ensure that **blitz.tcl** will be loaded when the command **blitzburp** is first used. The 0 indicates that the command must be typed exactly, and the 1 indicates that the interpreter will not evaluate arguments passed to the command. (See “Swat Reference,” Chapter 4, for more information on the **autoload** command.)

5.7.3 Explicit Loading

Another way to load a command into Swat is to use the **load** command from the Swat command line. This command is simply **load <path>/<filename>**. If no path is given, then the **<file>** is loaded from the directories specified in the **load-path** variable. The **load** command will load the given file (a Tcl procedure or subroutine) into Swat for subsequent use, and it is mostly used



Tool Command Language

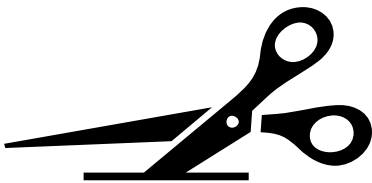
332

to load infrequently accessed files. (See “Swat Reference,” Chapter 4, for more information on the **load** command.)

5.7



Debug Utility



6.1	Changing Platforms.....	335
6.2	Switching Kernels.....	336

6





The Debug application allows the user to quickly change the debugging environment on the target machine. It works with the target's .ini to simulate the UI of different kinds of machines. It also allows for an easy way to switch back and forth between the regular and profiling kernels.



Warning:

Debug's simulation simulates the UI layouts of various platforms. While this is useful for laying out an application's UI gadgetry, remember that the real platforms will have different sets of system files and different hardware. For more complete testing, you should use an environment with the complete system files (e.g., to test a Zoomer program, use the ZOOM target directory). If your program is meant to work on a particular type of device, you should run some tests on that device as well.

6.1

6.1 Changing Platforms

Select the "Platform" item of the Options menu to change to another platform. Debug allows the target machine to simulate different platforms by means of .ini files which are stored in the target machine's INI subdirectory of the top-level GEOS directory. Depending on the contents of the selected .ini file, GEOS will act like a different device.

To select a new .ini file, click on it. A description of the platform simulated by the .ini file will appear. Some .ini files have been set up to simulate special hardware devices; others have been set up to test performance under different video modes. When you have selected the .ini file you want, click on the OK button to shut down GEOS and restart using the new .ini file.

Note that your personal .ini file is preserved—its [paths] *ini* field is updated to link in your selected platform .ini file.

If you wish to set up your own platform .ini file, make sure that it is in the proper subdirectory and contains a description of what it is simulating. Debug looks for .ini files in the INI subdirectory of the top-level GEOS directory. The platform descriptions are stored in the *notes* field of the [system] category in the .ini file. See one of the provided .ini files for an example of this.

6.2 Switching Kernels

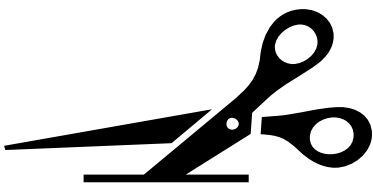
The “Profile” item of the Options menu allows you to switch between the regular and profiling GEOS kernels. The profiling kernel runs slower than the regular kernel, but compiles information useful for optimizing geodes.

When debugging, you may have several kernels residing in your SYSTEM directory. Debug creates a batch file which copies the appropriate kernel to geos.geo (or geosec.geo on an error-checking system). This kernel will then be used when GEOS restarts.

6.2



Icon Editor



7.1	Creating Icons.....	339
7.2	Importing Icons.....	340
7.3	Editing Icons	340
7.4	Writing Source Code	341
7.5	Icon Databases	342
7.6	Exporting to Database.....	343

7





The Icon Editor tool allows you to create icons for your application, both file icons used by file manager applications and simple tool monikers which you may use to provide graphic monikers for a geode's UI gadgetry.

7.1 Creating Icons

7.1

The icons produced by the icon editor are stored in an "icon database" so that they can be viewed and retrieved easily. The icon database is the Icon Editor's "document", so New/Open, Close, Save As, Make Backup and other Document-control commands will operate on the current database. "Save" actually saves the currently edited icon into the current icon database, and saves the database as well.

There are 3 standard kinds of icons in use in the system. Each standard type of icon has 3 default formats: One for VGA screens, one for MCGA (monochrome VGA) screens, and one for CGA screens.

Most icons are used in one of the following ways:

- ◆ File Icons, which are used by GeoManager to display and launch applications. The sizes are 48x30, 48x30 (monochrome), 48x14, 32x20, and 32x20(monochrome).
Note: The icon editor does not automatically create the 32x20 icon formats. You will need to do this yourself, using the Add Format dialog available under the Format menu.
- ◆ Tool Icons, which are used by controllers in their toolbox UI. The format sizes are 15x15 and 15x10.
- ◆ Mouse Pointers, which are used to provide a cursor tracking the mouse's movement. This format has one standard size:16 x 16.

You may also create custom icons, which start with 1 default format. Custom icons are limited to 1024 pixels wide or tall, and the formats cannot be larger than 64k.

The first time the icon editor is started it will create a blank file icon named "untitled." After that, if there is an icon database available to the icon editor,



the icon editor will start with the last icon that was being edited before the editor was shut down.

To create a new icon, choose “Create New Icon” from the Icon menu. The dialog will present 4 options for the type of icon. If you decide to create a custom icon, you should enter the height, width and color scheme for the icon in the provided fields.

7.2

7.2 Importing Icons

There are 2 other ways to create icons. You can import a graphic such as a GIF, TIF, BMP or other supported file format for use as an icon, provided that it is less than 64k (which is the limit for GEOS monikers). You do this using the “Import Graphic” dialog in the Icon menu.

The other way to make an icon in the icon database is to import a moniker or moniker group from the token database, which is the cache of file monikers kept by GeoManager. Select “Token Database” from the Icon menu to get a list of monikers that you may import.

7.3 Editing Icons

There are a number of features that are useful when editing icons:

- ◆ You can create a format for the icon, and scale it to help create the other formats, using the “Transform Format” dialog under the Format menu.
- ◆ The pixel-edit size can be set using the Options menu. It defaults to 8-bit-wide pixels.
- ◆ The “Preview” dialog allows you to view a format in a generic object. If the format is intended for use in a generic object, you can see what it looks like inverted, and (in the case of triggers and tools) what it looks like with different background colors.
- ◆ You can add your own formats, or delete existing ones, using commands from the Format menu. You might, for example, want to add a SuperVGA format for the icon (system supervga file-icons are 64x40).



- ◆ 256-color icons are technically supported, but no special tools are provided for editing them. You can select the red, green, and blue components of the area or line color from the “Area Color” and “Line Color” dialogs. 256-color icons aren’t much use unless either:

They use colors only from the system’s default palette, or

They are being used in a 24-bit color environment.

- ◆ You can select the aspect ratio for your icon. This will allow the system to make the best choice among formats when selecting from your moniker list. The choices are CGA, EGA, and VGA. The “VGA” aspect ratio should be used for VGA, MCGA (monochrome VGA), and SVGA formats.

7.4

- ◆ You are not required to create multiple formats for your artwork; however we recommend that if you are creating art that could be used under many video modes that you create the three standard formats for the art (VGA, mono VGA, CGA). This shouldn’t be too difficult, since there are several tools for working with several formats:

The “transform format” dialog can be used to create exact copies, or transformed copies, of one format to another.

The “resize format” dialog changes the size and shape of a single format.

7.4 Writing Source Code

When the icon is finished, make sure it’s saved (File->Save), and then choose “Write Source Code” from the Icon Menu. Follow these steps to get source code:

- 1** Enter a DOS 8.3 filename for the source file
- 2** To write only the current format, leave the “Formats” group at “Current Format”
- 3** To write source for all the formats, choose “All Formats”
- 4** Select GOC source (assembler is used in-house)

A dialog should appear when the source code has been written, saying “Source Code Written Successfully”. If a problem is encountered in creating



or writing a file, check the permissions on your document directory, and make sure you have enough disk space for the text file (only a few hundred bytes should be necessary).

The file should appear in your document directory. It's a text file with GOC moniker source code. This source code can be included in your application after you make the following changes:

7.5

- ◆ The formats for the icon will have names like "Moniker0". You should give them appropriate names.
- ◆ The monikers must be placed in a resource. You could place them between statements like:

```
@start APPMONIKERRESOURCE, data;  
@end APPMONIKERRESOURCE;
```

In your .gp file, this resource should have the attributes "lmem, read-only, shared".

- ◆ You may wish to add the line "style = tool;" for icons that are going to be used as toolbox UI, to assist the specific UI in deciding how to draw them.
- ◆ Your .goc file must #include <gstring.h>, an include file in which the graphics string data structures used to describe visual monikers are described. (Esp programs include **gstring.def**.)

7.5 Icon Databases

The icon editor can work with more than one icon database at a time via the Multiple Document Interface standard. Each database has its own window, each window having four views. The leftmost view is the database. The top-right view is the "Pixel View", the main editing area. The two remaining views are next to each other in the lower-right portion of the window. The left view is the actual-size view of the icon, and is editable. The right window is the format list, showing all formats of the icon currently being edited.

Double-clicking an icon in the database viewer will select it for editing. The currently-edited icon is important in the following ways:

- ◆ It is (of course) the only icon being edited in the database associated with that window.



- ◆ Its size determines the shapes of the various views, which will try to adjust themselves to fit in the window as best they can.
- ◆ It is the icon that appears in the “Preview” dialog.
- ◆ It is the icon used in the “Export to Token database” dialog (explained below).
- ◆ It is the target of all operations in the Icon and Format menus (except “create new icon”).

7.6

The database viewer supports all clipboard operations except for Undo. Icons may be selected and moved or copied between databases. This can be done via the Edit menu or quick-transfer.

The format list is used to switch editing to a different format in the same icon. This can only be done with the mouse.

To get more room for editing, use the Options menu to temporarily hide the database viewer or format list while editing.

7.6 Exporting to Database

This is the function that most people associate with an icon editor. the token database is a cache of moniker lists used by GeoManager and other applications. The token database exists so that every time an application's moniker is needed, it is not necessary to start the application in engine mode to request the moniker (a slow process).

The icon editor can change entries in the token database. You cannot change the “built in” moniker for a geode using the icon editor. If you make changes to the token database and then delete the database, then when it gets rebuilt your monikers will no longer be there.

Given that, it's still likely that people will want to set their own monikers in the token database; the icon editor has this ability.

To change an entry:

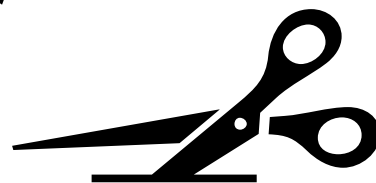
- ◆ Make sure you are editing the icon you wish to change.
- ◆ Open the File->Export to Token Database dialog.



- ◆ One of the icon's formats should be visible under the "Change To:" text in the dialog. If not, then you are probably editing the wrong icon, or perhaps the icon hasn't been saved into the database.
- ◆ Use the file selector to find the application, document, library, or other driver whose token database entry you wish to change. Otherwise, enter the four token characters for the geode into the Token Characters text field. Select the corresponding operation from the list at the lower-right of the dialog ("Use File Selector" or "Use Token Chars.")
- ◆ The current token-db entry for the selected geode should appear under the "Current Icon" title at the upper-right corner of the dialog.
- ◆ When you are satisfied with the edited icon, change the moniker with the "Change" trigger in the dialog's reply bar. The change will not take effect in GeoManager until you select Disk->Rescan Drives in GeoManager.
- ◆ If you wish to revert a token-database entry to the original, built-in moniker for the geode, select the geode in the file selector and use the "Remove" trigger in the reply bar.



Resource Editor



8

8.1	Glossary	347
8.2	Getting Started	350
8.3	What Needs to be Translated?	350
8.4	Translating	350
8.4.1	Choosing a new translation file	351
8.4.2	Main translation screen.....	351
8.4.3	Translating a Text String	352
8.4.4	Moving between chunks	353
8.4.5	Moving between resources	353
8.5	Resource Editor Menus.....	354
8.5.1	File Menu	354
8.5.2	Edit Menu	355
8.5.3	Project Menu.....	355
8.5.4	Filter Menu.....	356
8.5.5	Utilities Menu	356
8.5.6	Window Menu.....	357
8.6	Creating an Executable	357
8.7	Updating an Executable	358
8.8	Testing Your New Executables	359





The Resource Editor is a tool created by Geoworks to translate the existing English language GEOS executables into your local language.

As with any other application, the best way to learn how to use this tool is by experimentation. As you work through these instructions, have a copy of the Resource Editor running on one of your machines so that you can try for yourself some of the tasks outlined here.

8.1

Note that this documentation is somewhat redundant with other parts of the GEOS Technical Documentation. This is because we've tried to make ResEdit usable by non-programmers. Ideally, the only skills one should need to localize an application to a foreign language is knowledge of the source and target languages—the translator shouldn't have to work with the source code. If you are following this model, you may wish to give just this piece of documentation to your translator instead of burdening them with a full documentation set.

8.1 Glossary

In explaining how to use the Resource Editor, it is important that we use the same vocabulary to describe a certain portion of the code or the operation which must take place. In order to alleviate confusion, take a look through the following definitions, and refer back to them as necessary as you read the rest of these instructions:

Executable (Geode)

An executable (also called a geode) is a combination of assembly code that may be run and strings or bitmaps which may be displayed. Some executables, such as applications, are run by the user, while others, like printer drivers, are loaded and run automatically by GEOS when the user specifies an action like printing.

Source Executable

The original English language executable.



Destination Executable

The local language executable, which is created by translating the source executable.

Resource

Geodes consist of a series of divisions known as resources. Resources may contain code that will be executed, data that will be referred to, or strings/bitmaps which will be displayed to users. GEOS uses the concept of resources to minimize the amount of memory an application uses by only loading into memory those resources needed. The Resource Editor will only allow you to edit those resources which contain strings or bitmaps. Each of these will have its own unique number or extended name.

Chunk

Chunks (or more formally known as local memory chunks) are yet another way GEOS divides up an executable. Inside a resource, there are many different kind of chunks, each containing a unique piece of information, such as a text string, a bitmap or an object structure. All executables have their externally visible strings stored in separate chunks, which allows you to modify those chunks and create a translation. The Resource Editor will only allow you to edit chunks, not the lower level of bits and bytes. Each chunk will also have a unique number or name.

Object

The GEOS operating system is object oriented, that is, the visible items on your screen - windows, triggers, dialog boxes - are all objects which are stored in local memory chunks. Each object may have a moniker and some objects have keyboard accelerators. These two attributes are editable.

Driver

A driver is a special sort of executable that performs a very specific function that may not be needed by every user. In GEOS, there are DOS, video, mouse, and printer drivers, all created to minimize the amount of memory used by the system. Most drivers are designed to work with more than one device. For example, the EPSON9.GEO printer driver actually supports more than 50 printers, and in fact, the names of these printers are exactly what you will be able to translate.

Localization File

When a geode is created, an extra file to be used by the Resource Editor is created at the same time. This file (which is named like the .geo file but with a .vm extension) contains



localization information about the resources and chunks which you will be editing. You cannot edit a geode without this file.

Translation File

The file that the Resource Editor creates is called a translation file. The information stored in this file will be merged with the source executable to create the destination executable.

Internally, the file holds a copy of the original English text or bitmap, along with your translation and may contain some translation instructions.

8.1

Text String Longer text such as error messages.

Moniker Shorter text which serves as the visual name for objects such as menu items, button labels, and dialog box options.

Mnemonic A key that when pressed with the Alt key will activate a specific menu item which has this letter or symbol underlined in its moniker.

GString Graphics string (not editable with current Resource Editor)

Bitmap Actual bitmap image (not editable with current Resource Editor)

Keyboard Shortcut

A combination of keystrokes than when pressed will cause an object to perform a specific action. Keyboard shortcuts are stored in the chunk containing the object to which they are connected.

Updating Unfortunately, software changes. As you make changes to the English language software, it might become necessary for you to take a translation file you created with an older version of a particular geode, and update it with the newer version. The newest software may contain new strings, have modified existing strings, or eliminated some strings. The Resource Editor is however able to match these two executables and show the newest changes. The process of updating a file will be discussed in more detail later in these instructions.

8.2 Getting Started

You will need to install localization target version of Ensemble 2.0 in a directory of your target machine (for example \TARGET.20); This environment will be your destination for new localized executables. As you work along, you will replace the English executables in this structure with your translated executables. This will be explained in more detail later.

8.2

Note: you may wish to not have these directories in your path. We suggest that you edit batch files to run each installation.

8.3 What Needs to be Translated?

To create a localized version of your applications, you will need to translate only the applications, libraries, and drivers you have created. All Geoworks software will be translated by Geoworks.

Most applications will use GEOS system libraries whose own UI is visible from the application. For example, an application which includes spell checking will have a dialog box containing all the information about the spell checking operation which is included from the Spell library. If the Spell library is not translated, that UI will always appear in its original form. Your application will work with the Spell library, whether or not it has been translated.

8.4 Translating

At this point you are ready to translate geodes into the target language. The following set of steps can be carried out by a non-programmer running the



ResEdit tool. Of course, the translator should have a good idea of the geode's function and use.

8.4.1 Choosing a new translation file

To begin creating new translation files, double click on the “Res Edit” icon in the Tools folder of the WORLD directory of the LOCALIZE.20 version of Ensemble 2.0.

8.4

This brings up a dialog box which gives you two choices:

- ◆ Create New Translation File
- ◆ Open a Translation File

Choosing the “new” option opens a second dialog box which asks you to select the localization (.vm) file for the geode you wish to edit. Choosing the “open” option brings up an existing translation file for further editing. In either case, the order in which you do the translations is your decision.

The first time you open the Resource Editor, it assumes you will be editing geodes in the GEOS20 directory, or whatever installation of GEOS you started in. Once you have a translation file open, you can then change the top-level directory by clicking on the “Set source directory” in the Project Menu. This setting is saved in your .ini file, so the next time you start ResEdit, it will automatically set the top-level source directory to what it was when exited.

If you want to create a new translation file for a geode that is not in the GEOS20 installation (or whichever installation is currently set as the top-level source directory) you must change the top-level source directory first.

8.4.2 Main translation screen

After selecting a localization (.vm) file to translate, the Resource Editor will then load the information from this file and the corresponding geode into an untitled Translation file saved into the DOCUMENT directory. The file will be displayed in two views. The left portion of the screen is the Source File, while

the right portion is the Translation File. When the file is first opened, the two sides look the same, since no translations have yet been made.

The Resource Editor brings up all the strings or bitmaps which need to be translated, regardless of their context. The only way you will be able to be completely sure that the translation is accurate is to enter the translation and then actually run the translated executable (to be discussed later) and see how the translations appear in the program.

8.4

8.4.3 Translating a Text String

Click on any text string in the right portion of the screen. The blinking text editing cursor should appear, and you will be able to edit the text in the right screen to reflect the changes necessary for the local language. The text in the left portion of the screen acts as a guide and will not change.

At the bottom of the main screen, there are three fields which may have important information about how you should translate the selected chunk.

- ◆ The Minimum and Maximum Length fields show the size limits on the number of characters that the selected chunk may have.
- ◆ The Instructions field shows special characteristics of the selected chunk, or may give a context where the specific chunk appears.

If there is nothing in these fields, there are no constraints on the size or content of the translation.

You will also notice that one letter of many words in the chunks is underlined. This “mnemonic” can be changed by using the two arrows in the upper right corner of the main translation screen labeled Mnemonic. Moving up or down will move the underscored letter to the left and right in the translated chunk. You can also edit a mnemonic by selecting the text in the Mnemonic display and replacing it with the new letter. The first occurrence of that letter in the text string will now be underlined. If that letter is not in the text, it will become parenthesized in the Mnemonic object. This type of mnemonic is displayed not as an underlined character in a moniker, but is drawn in parenthesis after the moniker text. This type of mnemonic is rarely used, but may be helpful if you have mnemonics which conflict. You must be careful not to assign the same mnemonic to two monikers whose objects are both in the active window at the same time. If there is such a conflict, only one of the two



objects will be activated by this mnemonic. When you have created the executable, you should check that there are no conflicting mnemonics, and change any which are found to overlap with others at the same level.

In some cases, an object will have a keyboard shortcut in addition to a moniker mnemonic. These keyboard shortcuts consists of a combination of control keys and character keys or can be function keys (F1-F12). If you see "Type : Chunk" in the upper right corner, the currently highlighted chunk will contain a non-editable textual representation of the object's keyboard shortcut. The keyboard shortcut trigger will now be enabled (the Shortcut button in the middle of the main translation screen), and you can pop up a dialog box containing buttons and text which will be used to modify the shortcut. Currently, only text-based shortcuts are displayed, and are therefore, modifiable. Again, it is important to watch out for overlapping shortcuts, as these cannot be detected by the Resource Editor.

8.4

8.4.4 Moving between chunks

- ◆ Use the mouse to click on any chunk.
- ◆ Type Ctrl . (Ctrl-period) to move to the next chunk or Ctrl , (Ctrl-comma) to move to the previous chunk.
- ◆ Click on Next Chunk/Previous Chunk in the Utilities menu.
- ◆ Click on the Chunk pop-up menu in the upper left corner of the screen and choose any chunk.

8.4.5 Moving between resources

- ◆ Type Ctrl > (Ctrl-greater) to move to the next resource or Ctrl < (Ctrl-less) to move to the previous resource.
- ◆ Click on Next Resource/Previous Resource in the Utilities menu.
- ◆ Click on the Resource pop-up menu in the upper left corner of the screen and choose any resource.

8.5 Resource Editor Menus

Many of these features available in the Resource Editor are identical to those in the Ensemble 2.0 software. The quick overview of the menu items below should give you the necessary information to use all of the features.

8.5

8.5.1 File Menu

New/Open: brings up the New/Open dialog box, allowing you to open a new translation file or an existing one.

Close: closes the active translation file.

Save: saves the open translation file.

Save As: saves the open translation file under a different name.

Backup: makes a backup of the open translation file or restores from a backup file.

Other:

Copy To: creates a copy of the translation file and places it in a user selected directory.

Discard Changes:
discards all changes since last saved version of the translation file.

Rename: gives the translation file a new name.

Edit Document Notes:
edits the notes for the translation file.

Set Document Password:
creates a document password.

Set Document Type:
allows the user to choose between a normal, read-only, or public document.

New Name and User Notes:
allows user to change geode name and make annotations to the file for later reference.



Create a New Executable:

takes the translation file, merges it with the source code and creates a new localized executable. The process of creating executables will be discussed in more detail later in this document.

Update:

Update Translation file:

using a new or updated geode, matches chunks in the old file with those in the new file, creating an updated translation file which shows new or changed chunks and groups deleted chunks at the end.

8.5

Commit the update:

deletes intermediate matching information, showing only the newest version of the translation file

Exit: exits the Resource Editor

8.5.2 Edit Menu

Cut: takes highlighted information and moves it to the clipboard

Copy: takes highlighted information and copies it to the clipboard, leaving a copy in place

Paste: pastes information from the clipboard to the location of the cursor

Undo: reverts to previous saved version of individual chunk

Find and Replace:

allows user to search for designated words throughout the translation file and replace text in the translation file (as displayed in the right view) with local language translation. You can limit a forward or backward search by selecting filters for certain types of chunks from the Filters menu.

8.5.3 Project Menu

The items in this menu keep track of how your project is organized.

Source directory:

Set the top level directory holding source geodes.



Destination directory:

Set the top level directory holding modified geodes.

Reset Source Geode Path:

Reset the path of the source geode if it moves to a different subdirectory.

8.5.4 Filter Menu

8.5

Each of these menu items is a radio button, which allows you to turn on or off as many of these filters as you like. These options can be helpful when editing updated translation files.

Don't show Text:

Don't show Monikers:

Don't show GStrings:

Don't show Bitmaps:

Don't show Objects:

Each of these menu items is a radio button which allows you to view specific chunks affected by updating the translation file.

Show changed chunks:

Show new chunks:

Show deleted chunks:

8.5.5 Utilities Menu

These menu items allow for quick movement between chunks and resources.

Next Chunk:

Previous Chunk:

Next Resource:

Previous Resource:



8.5.6 Window Menu

These menu items make manipulation of multiple translation files easy, by opening overlapping windows or tiling active windows.

Overlapping:

Full-Sized:

Tile:

8.6

8.6 Creating an Executable

After all of the chunks and resources in the selected .vm file have been translated and saved in a translation file, a new local language executable can be created in the target installation.

The first step to placing the new executable in the correct place is making sure that the localization target kit has been placed in another directory on your hard drive, for example \TARGET.20.

Then select the Destination directory option from the Project menu. It will ask you to choose the top level GEOS directory that will hold the modified executables. In this case, you would choose the Path button, move to the C:\ directory, and choose \TARGET.20. You will need to do this each time restart the Resource Editor, since the destination path is not saved in your .INI file.

You need not have a full GEOS installation in the target directory. However, when new executables are created, they are placed into the same subdirectory as the original geode. For example, if you were to translate GeoWrite which is in the WORLD directory, and you had the destination directory set to TARGET20, the directory TARGET20\WORLD must exist when you try to create the new executable. If it doesn't you will get an error message saying that the target could not be created.

This new geode will have the identical English language long name as the original geode (unless you have changed the name in the File Menu's New Name and User Notes dialog) and will therefore overwrite the English file. However, you may wish to rename several of the geodes, especially those in the WORLD directory and the Screen Savers, to reflect local language names.



In this case, the new geode will not overwrite the original one, and you will have to go back and delete the old one to avoid duplicates.

8.7 Updating an Executable

8.7

Even after all the translations are complete, there may be a need to replace a certain executable with a newer version containing a bug fix or a feature enhancement. You will be able to do this quickly and effectively by using the Update feature in the File menu.

To have this feature work, you must first put the updated geode and .vm file in the proper directory, overwriting the original files. Then, by clicking Update Translation file (in the Update menu), the existing translation file will be compared with the new strings in the geode, showing the new or changed chunks, as well as the ones left unchanged. The deleted chunks will be grouped in a separate resource named “Deleted chunks” at the end of the resource list. You will then be able to make the necessary modifications to this file to bring it up to date with the newest version of the geode.

After updating the translation file, you can then create an updated local language executable as outlined above.

If it is clear that there have been no changes to the geode affecting editable chunks (as when code changes for a bug fix), it is also possible to simply rebuild the new geode from the existing translation file using the Create A New Executable feature in the File menu.

After viewing the updated translation file and making any necessary changes, you can remove the updating information from the translation file by clicking on the Commit the Update trigger in the File/Update menu. This will remove information which marks the chunks as “new” or “changed” and will delete the Deleted Chunks resource.



8.8 Testing Your New Executables

You may find that you would like to be able to test your translations as soon as you make them. At any point during a translation, you can create an executable.

Simply exit the LOCALIZE.20 directory, move to your destination directory (for example \TARGET.20) restart that installation of Ensemble 2.0 and test your new geode.

8.8

If your product will run under several video drivers (e.g. if it runs on desktop PCs), we suggest that you take a look at the localized UI under the CGA video driver to make sure that everything fits. You should also observe it under the EGA video mode with a large UI font size.

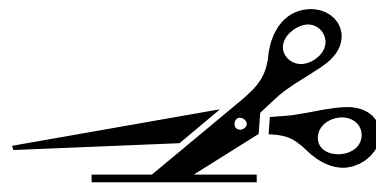
Resource Editor

360

8.8



The INI File



9

9.1	How to Use the INI File	363
9.2	Categories in the INI File	364
9.2.1	[cards]	370
9.2.2	[configure]	371
9.2.3	[diskswap]	372
9.2.4	[envelope]	373
9.2.5	[envel<num>]	374
9.2.6	[expressMenuControl]	374
9.2.7	[fileManager]	376
9.2.8	[input]	378
9.2.9	[keyboard]	382
9.2.10	[label]	384
9.2.11	[label<num>]	385
9.2.12	[link]	386
9.2.13	[localization]	387
9.2.14	[math]	387
9.2.15	[modem]	387
9.2.16	[<modem name>]	388
9.2.17	[mouse]	391
9.2.18	[net library]	392
9.2.19	[paper]	392
9.2.20	[paper<num>]	393
9.2.21	[parallel]	394
9.2.22	[paths]	394
9.2.23	[printer]	396
9.2.24	[<printer device name>]	397
9.2.25	[screen 0]	399
9.2.26	[serial]	401



9.2.27	[sound].....	402
9.2.28	[spool]	402
9.2.29	[system].....	403
9.2.30	[text].....	409
9.2.31	[ui].....	413
9.2.32	[<specific ui name>]	421
9.2.33	[ui features]	421
9.2.34	[welcome].....	428



Most programmers are familiar with the structure and function of an initialization file, or INI file. GEOS uses one or more INI files depending on the setup (network users may have two or more INI files; standalone users may have only one).

The INI file for GEOS describes the drivers, fonts, and other items installed on the system. It also contains other system configuration information, such as the specific UI expected and information about the type of display and input devices used. It may also be used by applications for storing their configuration information as set by the user.

9.1

9.1 How to Use the INI File

As a software developer, you will have two uses for the INI file. First, the INI file controls your system configuration to a certain extent. For example, if you are developing applications for a small-screen pen device, you should know the appropriate settings in the INI file to get your system to emulate such a device.

Second, you may need to have your application access the INI file to store or retrieve information. The kernel offers routines for just this purpose; these routines are detailed in Chapter 6 of the Concepts books.

This chapter describes the file itself as well as the categories and keys created and used by GEOS. Depending on the system you plan to develop for, you may need to set several different keys in certain combinations (e.g. setting certain keys emulates a Zoomer configuration). These special combinations are described in the special add-on documents that describe developing for each system. For more information, contact Geoworks Developer Support.



9.2 Categories in the INI File

Code Display 9-1 shows a short list, without description, of all the categories and keys available in the INI file. Following the display are explanations of each category, with a description of each key and the values you can set.

9.2 Code Display 9-1 The GEOS.INI File

```
; This is a listing of many of the categories and keys in the GEOS.INI file.
; Each category is described in full in the following sections of this
; chapter, along with the values you can set for each key and what they do.
; The categories and keys are listed alphabetically.

[cards]
deckdir = <directory containing deckfile>
deckfile = <full name of deck>

[configure]
drive <letter> = <drive number>
helpEditor = <Boolean>
numberWS = <maximum number of calculator worksheets>

[diskswap]
file = <path of swap file>
page = <size of swap page>
size = <size of swap file>

[envelope]
count = <number of user-defined size strings>
newSizes = <list of user-defined size strings>
order = <array of DefaultOrderEntry values>

[envel<num>]
name = <name string>
width = <width in points>
height = <height in points>
layout = <PageLayout structure>

[expressMenuControl]
floatingKeyboard = <Boolean>
otherAppSubMenu = <Boolean>

[fileManager]
dosAssociations = {<list of associations>}
dosLaunchers = <Boolean>
dosParameters = <Boolean>
```



```
filenameTokens = {<list of associations>}
fontID = <font ID of font in folder windows>
fontSize = <point size of folder window font>
options = <number>
startupDrivesLocation = <number>
```

```
[input]
blinkingCursor = <Boolean>
clickToType = <Boolean>
doubleClickTime = <number of ticks>
keyboardOnly = <Boolean>
left handed = <Boolean>
mouseAccelMultiplier = <number>
mouseAccelThreshold = <number>
noKeyboard = <Boolean>
numberOfMouseButtons = <number>
quickShutdownOnReset = <Boolean>
reboot on reset = <Boolean>
selectDisplaysMenu = <Boolean>
selectRaises = <Boolean>
```

```
[keyboard]
device = <full device name>
driver = <driver file name>
keyboardAltGr = <Boolean>
keyboardDoesLEDs = <Boolean>
keyboardShiftRelStr = <Boolean>
keyboardSwapCtrl = <Boolean>
keyboardTypematic = <number>
```

```
[label]
count = <number of user-defined size strings>
newSizes = <list of user-defined size strings>
order = <array of DefaultOrderEntry values>
```

```
[label<num>]
name = <name string>
width = <width in points>
height = <height in points>
layout = <PageLayout structure>
```

```
[link]
name = <machine name>
port = <number>
baudRate = <number>
drives = <list of drives>
```

9.2

The INI File

366

9.2

```
[localization]
currencyDigits = <number of decimal digits for currency>
currencyLeadingZero = <Boolean>
currencySymbol = <character of currency symbol>
day
decimalDigits
decimalSeparator
hoursMins24HourTime
hoursMinsSecs24HourTime
hoursMinsSecsTime
hoursMinsTime
hoursTime
longCondensedDate
longDate
longDateNoWeekday
measurementSystem
minsSecsTime
month
monthDayLongDate
monthDayLongDateNoWeekday
monthDayShort
monthYearLong
monthYearShort
negativeSignBeforeNumber = <Boolean>
negativeSignBeforeSymbol = <Boolean>
quotes
shortDate
spaceAroundSymbol = <Boolean>
symbolBeforeNumber = <Boolean>
useNegativeSign = <Boolean>
weekday
year
zeroPaddedShortDate

[math]
coprocessor = <library name for coprocessor>

[modem]
modems = {<modem name list>}
numberOfModems = <number>

[<modem name>]
baudRate = <number>
handshake = <hardware, software>
parity = <none, even, odd, mark, space>
stopBits = <number>
```



```
stopLocal = <dsr, dcd, cts>
stopRemote = <dtr, rts>
toneDial = <Boolean>
wordLength = <number>

[mouse]
device = <full device name>
driver = <driver file name>
info = <number>
irq = <number>
port = <number>

[netLibrary]
    InitDrivers = {<list of driver geodes>}

[paper]
count = <number of user-defined size strings>
newSizes = <list of user-defined size strings>
order = <array of DefaultOrderEntry values>

[paper<num>]
name = <name string>
width = <width in points>
height = <height in points>
layout = <PageLayout structure>

[parallel]
port <number of parallel port> = <level of port>

[paths]
<standard path> = <other paths to merge>
ini = <additional .INI files to load>
inisaved = <path of saved .INI file>
sharedTokenDatabase = <path of shared token db file>

[printer]
count = <number>
defaultPrinter = <number>
numFacsimiles = <number>
numPrinters = <number>
printers = {<list of print devices>}

[<printer device name>]
baudRate = <speed of serial communication>
device = <full device name>
driver = <file name of driver>
handshake = <handshake for serial communication>
parity = <parity for serial communication>
```

9.2

The INI File

368

```
port = <port name>
stopBits = <stop bits for serial communication>
type = <type of print device>
wordLength = <word size for serial communication>
```

```
[screen 0]
device = <full name of device>
driver = <file name of driver>
oldDevice = <full name of device formerly used>
oldDriver = <file name of driver formerly used>
```

9.2

```
[serial]
port <number of serial port> = <level of port>
```

```
[sound]
sampleDriver = <driver file name>
synthDriver = <driver file name>
```

```
[spool]
uiOptions = <SpoolUIOptions>
```

```
[system]
continueSetup = <Boolean>
drive <letter> = <number>
font = <drivers to be loaded>
fontid = <font to be used as the default>
fontsize = <point size of default font>
fs = <drivers to be loaded>
handles = <number of handles>
inkTimeout = <ticks until ink is processed>
maxTotalHeapSpace = <memory size>
memory = <swap drivers to be loaded>
noFontDriver = <Boolean>
notes = <string>
noVidMem = <Boolean>
pda = <Boolean>
penBased = <Boolean>
power = <file name of power management driver>
serialNumber = <serial number of installed GEOS>
setupMode = <mode for graphical setup application>
splashcolor = <background color>
splashscreen = <Boolean>
splashtext = <text message>
```

```
[text]
autoCheckSelections = <Boolean>
autoSuggest = <Boolean>
dialect = <dialect code>
```



```
dictionary = <file name of dictionary used by spell checker>
hyphenationDictionary = <file name of dictionary>
hyphenationLanguage = <name of language>
language = <language code>
languageName = <name of language in use>
resetSkippedWordsWhenBoxCloses = <Boolean>
smartQuotes = <Boolean>
```

```
[ui]
autosave = <Boolean>
autosaveTime = <seconds between autosaves>
background = <file name of background graphic>
backgroundattr = <t, c, or x>
backgroundcolor = <color index of background>
confirmShutdown = <Boolean>
deleteStateFilesAfterCrash = <Boolean>
doNotDisplayResetBox = <Boolean>
execOnStartup = <list of programs to run on startup>
generic = <generic UI file name>
hardIconsLibrary = <string>
haveEnvironmentApp = <Boolean>
hwr = <file name of handwriting recognition library>
kbdAcceleratorMode = <Boolean>
noClipboard = <Boolean>
noSpooler = <Boolean>
noTaskSwitcher = <Boolean>
noTokenDatabase = <Boolean>
overstrikeMode = <Boolean>
password = <Boolean>
passwordText = <encrypted text>
penInputDisplayType = <number of display type>
productName = <name of the product>
screenBlanker = <Boolean>
screenBlankerTimeout = <number of minutes>
showTitleScreen = <Boolean>
sound = <Boolean>
specific = <specific UI file name>
tinyScreen = <Boolean>
unbuildControllers = <Boolean>
xScreenSize = <width of screen>
yScreenSize = <height of screen>

[<specific ui name>]
fontid = <font>
fontsize = <size in points>
```

9.2

The INI File

370

```
9.2 [ui features]
backupDir = <directory for quick backup files>
defaultLauncher = <relative path of application launcher>
docControlFSLevel = <number>
docControlOptions = <number>
expressOptions = <number>
helpOptions = <number>
interfaceLevel = <number>
interfaceOptions = <number>
launchLevel = <number>
launchModel = <number>
launchOptions = <number>
quitOnClose = <Boolean>
windowOptions = <number>

[uiFeatures - intro]
[uiFeatures - beginner]
[uiFeatures - advanced]

[welcome]
enteredprofessionalroom = <Boolean>
startup = <application name to start>
startupRoom = <name of startup room>
```

9.2.1 [cards]

The cards category contains information used by the cards library. The cards library provides routines used by card games. The cards category contains information about how to access the file containing the bitmap to be used as the card back picture.

deckdir

deckdir = <deck directory>

This optional field shows the path which contains the file named in the deckfile key, described below. If this key is not given, the cards library will look for deck files in the USERDATA\DECK directory.



deckfile

```
deckfile = <deck name>
```

The *deckfile* key defines the full name of the deck to be used by the card library. This is most useful in cases like the Zoomer, which must have its own card artwork. By default, the cards library will look in the USERDATA\DECK directory.

```
deckfile = Zoomer Default Deck
```

9.2

9.2.2 [configure]

The *configure* category contains miscellaneous configuration information for GEOS. The *drive* key is exactly like the *drive* key in the *system* category.

drive

```
drive <letter> = <number>
```

This key allows you to override the drive-map initialization done by the primary filesystem driver. You can not make a driver believe a nonexistent drive exists, but you can change the presumed media or make the driver ignore a drive. More than one drive may be remapped.

The *letter* argument is the drive letter of the drive to be remapped. The *number* argument defines the new drive definition and is one of the following values:

-1	fixed disk
0	ignore the drive
360	360 K 5.25-inch disk
720	720 K 3.5-inch disk
1200	1.2 meg 5.25-inch disk
1440	1.44 meg 3.5-inch disk
2880	2.88 meg 3.5-inch disk

Some examples of drive remappings are shown below:

```
drive d = 0      ; ignore drive D:
drive a = 360    ; make GEOS think drive A: is 360K
```



helpEditor

helpEditor = <Boolean>

If true, this key indicates that GeoWrite should add a new Help Editor feature. This must be on if you are planning on creating help files for your application. The Help Editor feature may be turned on by selecting “Fine Tune” in the user level dialog box in GeoWrite. More information on the Help Editor can be found in the chapter on the help system.

9.2

helpEditor = true

helpEditor = false

numberWS

numberWS = <number of calculator worksheets>

When users use the worksheets feature of the calculator, each worksheet they use will be loaded into memory. Normally, these worksheets are kept in memory so that the user may quickly go back to a worksheet they were using previously. Some low-memory devices may prefer that fewer worksheets are cached in this manner. The *numberWS* specifies a maximum number of worksheets that may be so cached.

9.2.3 [diskswap]

The *diskswap* category defines swap information for the GEOS swap file. Generally, you won't set these keys individually; GEOS will set them as required.

file

file = <path of swap file>

This category defines the file used by the disk swap driver for swapping.

file = C:\GEOWORKS\SWAP\EXTRA



page

page = <size of swap page>

This key defines the page size of a swap page.

page = 2048

size

size = <size of swap file>

9.2

This key defines the maximum size of the swap file.

size = 2048

9.2.4 [envelope]

This category keeps track of any customizations the user may have made to the list of envelope paper sizes.

count

count = <number>

This is the number of user-defined envelope sizes.

newSizes

newSizes = <list of paper size codes>

This list contains a list of all user-defined envelope sizes. The paper size information for each of these sizes will be stored in a category named [envelnum], where *num* is the three-letter code in this list.

order

order = <list of paper size codes>

This list contains a list of all envelope sizes in the order that the user wants them to appear in envelope size lists.



9.2.5 [envel<num>]

This category contains size and layout for a user-defined envelope size.

height

height = <number>

This field holds the envelope's height in points.

layout

layout = <PageLayout value>

This field holds the envelope's layout information.

name

name = <string>

This field holds the envelope's full name.

width

width = <number>

This field holds the envelope's width in points.

9.2.6 [expressMenuControl]

The *expressMenuControl* category defines the configuration of the express menu; see the *ui features* category for more express menu controls.

floatingKeyboard

floatingKeyboard = <Boolean>

If true, this key adds an item to the express menu to bring up the floating keyboard (used for pen-based systems). The default is false.

floatingKeyboard = true

floatingKeyboard = false



maxNumDirs

`maxNumDirs = <number>`

If this field exists, then if there are more entries in the Other Apps section of the Express Menu than this, the Other Apps section will be forced into a submenu (and forced into a subgroup if less than this), regardless of 'noSubMenus' and 'otherAppSubMenu'. Defaults to 25 (the absolute maximum number of Other Apps entries).

9.2

noSubMenus

`noSubMenus = <Boolean>`

If true, the express menu will not allow forcing the main applications, other applications, or desk accessories into submenus.

otherAppSubMenu

`otherAppSubMenu = <Boolean>`

If true, this key turns the "other applications" section in the express menu into a submenu rather than a subgroup. The default is false.

```
otherAppSubMenu = true
otherAppSubMenu = false
```

runningAppSubMenu

```
runningAppSubMenu = <Boolean>
```

If true, this key causes the express menu include a list of currently running GEOS applications as a submenu. If false, the list will be placed directly in the express menu.

runSubMenu

```
runSubMenu = <Boolean>
```

If true, top level applications (those placed in the WORLD directory) and top level subdirectories (subdirectories of WORLD) in a submenu of the express menu.

9.2.7 [fileManager]

The *fileManager* category is used by file manager applications such as GeoManager. You will probably not find much cause to use these keys during your application development.

dosAssociations

```
dosAssociations = {<list of associations>}
```

This key allows a user to associate DOS data files with DOS executables so a particular DOS executable will be launched when the user double-clicks the data file.

```
dosAssociations = {  
    *.ZIP = C:\PKUNZIP.EXE  
}
```



dosLaunchers

dosLaunchers = <Boolean>

If true, this key allows DOS launchers to launch DOS programs. The default is true.

```
dosLaunchers = true
dosLaunchers = false
```

9.2

dosParameters

dosParameters = <Booeans>

If true, the file manager will allow the passing of parameters to DOS executables..

filenameTokens

filenameTokens = {<list of associations>}

This key allows the user to set icon associations with DOS files. It also allows certain text files to be opened by the text file editor. Certain associations are made by default and should always appear; these are listed below.

```
filenameTokens = {
    *.EXE = "gDOS", 0
    *.COM = "gDOS", 0
    *.BAT = "gDOS", 0
    *.TXT = "FILE", 0, "TeEd", 0
    *.DOC = "FILE", 0, "TeEd", 0
    *.HLP = "FILE", 0, "TeEd", 0
}
```

fontID

fontID = <number>

This key sets the font used by GeoManager (or another file manager) when displaying the names and information of files in the folder window. The font

ID is set the same way as for the *system* category; if no font ID is named, the default system font will be used.

```
fondID = berkeley
```

fontSize

```
fontSize = <number>
```

9.2

This key sets the font size used in a folder window. If not specified, it will default to the system font.

```
fontSize = 10
```

options

```
options = <number>
```

This key controls various file manager options. The options are set and cleared by the user using the file manager's Options menu; the number is a decimal number representing the bits set or cleared for various options.

startupDrivesLocation

```
startupDrivesLocation = <number>
```

This key controls the initial location of drive buttons; it is set with the Options menu in the file manager.

9.2.8 [input]

The input category contains a number of keys that define how input is handled by the system. It affects the keyboard and mouse setup as well as how the UI responds to various user actions.



blinkingCursor

blinkingCursor = <Boolean>

If true, this key forces the text cursor to be a blinking cursor; it defaults to true. The screen dumper application requires a non-blinking cursor.

```
blinkingCursor = true
blinkingCursor = false
```

9.2

clickToType

clickToType = <Boolean>

If true, this key requires the user to click in a window before the focus will change to that window. When this key is false, the UI will invoke a “real estate” mode, wherein the user’s typing will go to the window under the mouse, whether the mouse was clicked there or not. The default is true (click required).

```
clickToType = true
clickToType = false
```

doubleClickTime

doubleClickTime = <number of ticks>

This field specifies the time threshold between clicks which should be recognized as a double-click. This time is expressed in 1/60th second “ticks”. The default value is 20.

keyboardOnly

keyboardOnly = <Boolean>

If true, this key indicates that GEOS is running on a system with only a keyboard (no mouse) for input. The default is false.

```
keyboardOnly = true
keyboardOnly = false
```



left handed

```
left handed = <Boolean>
```

If true, this key switches the left and right mouse button significance. For single-button mice, there is no effect; for three-button mice, the middle button stays the same. The default is false.

```
left handed = true
```

```
left handed = false
```

9.2

mouseAccelMultiplier

```
mouseAccelMultiplier = <number>
```

This key gives a multiplier to allow mouse acceleration along the following rule: any single event pixel movement beyond a given threshold (see *mouseAccelThreshold*) is multiplied by the multiplier. The multiplier defaults to one, which provides no acceleration. (A multiplier of three or four is fast.)

```
mouseAccelMultiplier = 1           ; no acceleration
```

```
mouseAccelMultiplier = 4           ; very fast
```

mouseAccelThreshold

```
mouseAccelThreshold = <number>
```

This key gives the mouse acceleration threshold used with *mouseAccelMultiplier* (above) to provide mouse acceleration. This threshold is the number of pixels the mouse must move before acceleration is invoked.

```
mouseAccelThreshold = 5
```

noKeyboard

```
noKeyboard = <Boolean>
```

If true, this key indicates that the system running GEOS has no keyboard. The default is false.

```
noKeyboard = true
```

```
noKeyboard = false
```



numberOfMouseButtons

numberOfMouseButtons = <number>

This key defines the number of buttons the mouse has. This may be the value one, two, or three. The default is three.

```
numberOfMouseButtons = 1
numberOfMouseButtons = 2
numberOfMouseButtons = 3
```

9.2

quickShutdownOnReset

quickShutdownOnReset = <Boolean>

If true, this key forces a Ctrl-Alt-Del sequence to force a dirty shutdown of GEOS. This defaults to true.

```
quickShutdownOnReset = true
quickShutdownOnReset = false
```

reboot on reset

reboot on reset = <Boolean>

If true, this key causes a Ctrl-Alt-Del sequence to warm-boot the machine rather than exit quickly to DOS. The default is false.

```
reboot on reset = true
reboot on reset = false
```

selectDisplaysMenu

selectDisplaysMenu = <Boolean>

If true, this key reverses the left and right mouse buttons with respect to menus in some specific UIs. (For example, the left (select) button will open the menu, but the right (features) button will show and execute the default menu item.) The default is false.

```
selectDisplaysMenu = true
selectDisplaysMenu = false
```



selectRaises

`selectRaises = <Boolean>`

If true, this key causes a select-button click to raise the window in which the click occurred, if the window was behind other windows. The window will not be raised above other windows that are always kept on top (e.g. the help window and desk accessory applications). The default is true.

9.2

```
selectRaises = true
selectRaises = false
```

9.2.9 [keyboard]

device

`device = <full device name>`

This key defines the keyboard in use. It must be the keyboard device's full name; in general, this should only be set by the Preferences manager application.

```
device = U.S. Keyboard
```

driver

`driver = <driver file name>`

This key goes with the device key and defines the driver file name to be loaded. This should be set by the Preferences manager.

```
driver = kbd.geo
```



keyboardTypematic

`keyboardTypematic = <number>`

This key defines both the repeat speed and delay before repeat for the keyboard. The number is an integer less than 128 (the high bit is ignored), and it is interpreted as three separate fields, as below:

bit 7	ignored	
bit 6-5	DELAY (see below)	9.2
bit 4-3	PE (exponent portion of repeat period)	
bit 2-0	PM (mantissa portion of repeat period)	

The delay is calculated by the following formula:

$\text{delay} = 1 \text{ second} + (\text{DELAY} * 250 \text{ ms}) \pm 20\%$

The period is calculated by the following formula

$\text{period} = (8 + \text{PM}) * (2^{\text{PE}}) * 0.00417 \text{ seconds}$

If no typematic number is specified, GEOS sets the default to 44, which represents a medium delay and a medium repeat period.

```
keyboardTypematic = 0    ; short delay, fast repeat
keyboardTypematic = 127 ; long delay, slow repeat
```

keyboardDoesLEDs

`keyboardDoesLEDs = <Boolean>`

If true, this key tells GEOS that the XT-class machine it's running on supports BIOS-updated LEDs for Num Lock, Caps Lock, and Scroll Lock. Most XT-class machines do not support updating these LEDs. This field is unnecessary on AT-class and more advanced machines.

```
keyboardDoesLEDs = true
keyboardDoesLEDs = false
```

keyboardAltGr

keyboardAltGr = <Boolean>

If true, this key makes the right Alt key function like Ctrl-Alt, as with many European setups.

keyboardAltGr = true

keyboardAltGr = false

9.2

keyboardShiftRelStr

keyboardShiftRelStr = <Boolean>

If true, this key makes the Shift keys release the Caps Lock, as with typewriters.

keyboardShiftRelStr = true

keyboardShiftRelStr = false

keyboardSwapCtrl

keyboardSwapCtrl = <Boolean>

If true, this key swaps the left Ctrl key with the Caps Lock key so the keyboard acts like many non-PC keyboards.

keyboardSwapCtrl = true

keyboardSwapCtrl = false

9.2.10 [label]

This category keeps track of any customizations the user may have made to the list of label paper sizes.

count

count = <number>

This is the number of user-defined label sizes.



newSizes

newSizes = <list of paper size codes>

This list contains a list of all user-defined label sizes. The paper size information for each of these sizes will be stored in a category named [label*num*], where *num* is the three-letter code in this list.

order

9.2

order = <list of paper size codes>

This list contains a list of all label sizes in the order that the user wants them to appear in label size lists.

9.2.11 [label<*num*>]

This category contains size and layout for a user-defined label size.

height

height = <number>

This field holds the label's height in points.

layout

layout = <PageLayout value>

This field holds the label's layout information.

name

name = <string>

This field holds the label's full name.

width

width = <number>

This field holds the label's width in points.



9.2.12 [link]

These fields are used by the Remote File System Driver to describe the machine to other machines that wish to access its drives. You may specify a name for the machine by which others may identify it and also set up communications parameters.

baudRate

9.2

```
baudRate = <number>
```

This **SerialBaud** value defines the communication speed this machine supports for RFSD.

drives

```
drives = <list of drives>
```

Normally, all of your drives will be accessible by the remote machine. This field allows you to specify exactly which drives are accessible. For instance, to restrict remote machines to accessing your C: and E: drives, use the following entry:

```
drives = {  
  C: E:  
}
```

name

```
name = <machine name>
```

A string by which remote machines may identify you. When the remote machine sees your drives, their names will be <machine-name>-<drive-letter>:.

port

```
port = <SerialPortNum>
```

This number specifies which serial port is to be used for RFSD.



9.2.13 [localization]

The *localization* key defines various configuration aspects of the system as set by the user in the Preferences manager application. Each of the keys in this category specifies one aspect of the user's system, typically an aspect defined by the country the user lives in. Because all of these keys are set in the Preferences manager application, they are not listed here. In general, keys which encode characters or strings will do so using ASCII values (e.g. "decimalSeparator = 2E" means that '.' is the decimal separator). The kernel also provides a number of routines to get and set the localization parameters; for more information, see the chapter on Localization in the Concepts books.

9.2

9.2.14 [math]

```
coprocessor = <driver name>
```

The *math* category allows the user to override the way GEOS normally treats math coprocessors. The single key (*coprocessor*) specifies the coprocessor library to use; the math library will load that particular library, and the library will check to ensure the proper coprocessor chip exists. If the chip is present, the library will be used; if the chip is absent, the math library will use software emulation of a coprocessor.

```
coprocessor = none           ; use software emulation
coprocessor = intx87.geo      ; Intel 80387, 80486
coprocessor = intx8087.geo    ; Intel 80287, 8087
```

9.2.15 [modem]

The *modem* category defines the modems attached to the system. Each modem must have its name listed in the name list, and each modem named in the list must have its own category (see the *modem name* category, below).

modems

```
modems = {<modem name list>}
```

This key defines all the modems attached to the system. Each modem in the list must have its own category in the .INI file, as shown below.

```
modems = My Modem
modems = { My Slow Modem
           My Fast Modem }
```

9.2

numberOfModems

```
numberOfModems = <number>
```

This key defines the number of modems specified in the *modems* list (above).

```
numberOfModems = 1
numberOfModems = 2
```

9.2.16 [<modem name>]

Each modem listed in the *modems* keyword in the *modem* category must have its own category. The category is named for the modem name in the list. Thus, the following example shows that each modem in the system has its own category:

```
[modem]
numberOfModems = 2
modems = {My Slow Modem
          My Fast Modem }

[My Slow Modem]
port = COM1
baudRate = 300
toneDial = true
parity = none
wordLength = 8
stopBits = 1
handshake = software
```



```
[My Fast Modem]
port = COM3
baudRate = 19200
toneDial = true
parity = none
wordLength = 8
stopBits = 1
handshake = software
```

9.2

baudRate

baudRate = <number>

This key defines the modem's baud rate.

```
baudRate = 2400
baudRate = 9600
```

handshake

handshake = <hardware, software>

This key indicates the type of handshake used by the modem. It must be one of the values specified.

```
handshake = hardware
handshake = software
```

parity

parity = <none, even, odd, mark, space>

This key indicates the modem's parity type. It must be one of the values shown above.

```
parity = none
parity = even
```

stopBits

```
stopBits = <number>
```

This key specifies the number of stop bits used by the modem. This should be 1, 1.5, or 2.

```
stopBits = 1
stopBits = 1.5
stopBits = 2
```

9.2

stopLocal

```
stopLocal = <dtr, dcd, cts>
```

If hardware handshaking is used, this key specifies which line the serial driver will watch for the stop signal.

```
stopLocal = dtr
stopLocal = dcd
stopLocal = cts
```

stopRemote

```
stopRemote = <dtr, rts>
```

If hardware handshaking is used, this key specifies which line the serial driver will use to make the remote side of the connection stop.

```
stopRemote = dtr
stopRemote = rts
```

toneDial

```
toneDial = <Boolean>
```

If true, this key indicates that the modem may use tone dialing. The default is true.

```
toneDial = true
toneDial = false
```



wordLength

`wordLength = <number>`

This key indicates the communication word length. This should be a number between five and eight inclusive.

`wordLength = 8`

9.2.17 [mouse]

9.2

The *mouse* category defines the mouse driver and specifics of the mouse attached to the GEOS system. Both the *device* and *driver* keys are required; the others are optional.

device

`device = <device name>`

This key defines the type of mouse attached. It must be the full device name and is typically set during setup of the system.

`device = Logitech Bus Mouse`

`device = No idea`

driver

`driver = <file name>`

This key defines the file name of the mouse driver in use.

`logibus.geo`

`logibuse.geo`

`genmouse.geo`

info

```
info = <number>
```

This key defines the “extra word” of data for the mouse. This is an internal structure set by the mouse driver.

irq

9.2

```
irq = <number>
```

This key allows you to set the interrupt level of a mouse that needs it; most mice do not need to be told their interrupt level.

```
irq = 4
```

port

```
port = <number>
```

This key specifies the port of a serial mouse, if necessary. The port number is one, two, three, or four, appropriate to the COM port being used.

```
port = 3
```

9.2.18 [net library]

```
InitDrivers = { <list of driver geodes> }
```

This is a list of network drivers to use. The Net library will attempt to load these driver geodes.

9.2.19 [paper]

This category keeps track of any customizations the user may have made to the list of paper sizes.



count

count = <number>

This is the number of user-defined paper sizes.

newSizes

newSizes = <list of paper size codes>

This list contains a list of all user-defined paper sizes. The paper size information for each of these sizes will be stored in a category named [paper*num*], where *num* is the three-letter code in this list.

9.2

order

order = <list of paper size codes>

This list contains a list of all paper sizes in the order that the user wants them to appear in paper size lists.

9.2.20 [paper<num>]

This category contains size and layout for a user-defined paper size.

height

height = <number>

This field holds the page's height in points.

layout

layout = <PageLayout value>

This field holds the paper's layout information.

name

name = <string>

This field holds the paper's full name.



width

`width = <number>`

This field holds the paper's width in points.

9.2.21 [parallel]

9.2

`port <number> = <level>`

The *parallel* category defines all the parallel ports available on the machine running GEOS. If a port is not defined in this category, GEOS will not recognize its existence. The single key in this category may be used numerous times, once for each available parallel port.

The *port* key defines the hardware interrupt level for the specified port. The normal entries for the three base parallel ports are shown in the examples below. If no level is provided for a port, the parallel driver will assign the values shown below; if no interrupt level is available for a port, GEOS will instead spawn a background thread for it. Setting any port's value will override other defaults (e.g. if you set port two to have level seven, port one will be set to level five, unless GEOS is on an XT- or PC-class machine).

```
port 1 = 7
port 2 = 5
port 3 = 0
```

9.2.22 [paths]

The *paths* category defines other directories to add into a standard path as well as additional INI files and the location of the shared token database. Adding directories to standard paths is useful both in network situations and if you want your application installed in a special directory but linked to the WORLD directory. This category uses five different keys, as shown below.



standard paths

<standard path> = <additional paths>

Each standard path is its own key, and you can merge other directories into any standard path. Some examples are shown below.

```
top = C:\GEOWORKS C:\PCGEOS
world = E:\INSTALL\NEWAPP
userdata font = N:\NETFONTS F:\SPECFONT
```

9.2

ini

ini = <file names>

This key defines up to three additional INI files to be loaded read-only. When GEOS searches for a key, the local (current) INI file is scanned first, followed by the additional INI files in the order they're defined. The first occasion of the key will be used; thus, the local INI file can supersede other settings.

```
ini = personal.ini INI\mydevice.ini n:\shared.ini
```

inisaved

inisaved = <files>

This key is used only if GEOS is run with the */psaved* argument. That is, if the user runs GEOS thus:

```
C>GEOS /psaved
```

then GEOS will look for the *inisaved* key rather than the *ini* key for additional INI files. Similarly, if the user instead runs

```
C>GEOS /pxxx
```

GEOS will look for a key called *inixxx* for the names of the files to be used.

```
inisaved = net.ini
inisaved = demo.ini net.ini
```

sharedTokenDatabase

```
sharedTokenDatabase = <path>
```

This key defines the location of the shared token database file. This key is most useful in network situations, when many users may be sharing a single token database.

```
sharedTokenDatabase = N:\NETFILES\TOKEN_DA.000
```

9.2

9.2.23 [printer]

The *printer* category defines all the printers configured for the system running GEOS. The *printers* key within this category (see below) defines the printer names, each of which must then have its own category in the INI file (see the following section under *printer device name*).

count

```
count = <number>
```

This key indicates the number of printers installed.

```
count = 0           ; no printers installed
count = 2           ; two printers installed
```

defaultPrinter

```
defaultPrinter = <number>
```

This key specifies the number of the installed printer that will act as the default device.

```
defaultPrinter = 2      ; printer #2 is the default
```

numFacsimiles

```
numFacsimiles = <number>
```

This key specifies the number of installed print devices which are actually fax drivers rather than printer drivers.

```
numFacsimiles = 1
```



numPrinters

```
numPrinters = <number>
```

This key specifies the number of installed print devices which are actually printers (as opposed to faxes or other devices).

```
numPrinters = 2
```

printers

9.2

```
printers = {<list of devices>}
```

This key defines all the installed print devices. Each entry in the list is the device name; the list must be a blob, with one printer named per line as in the example below. Each entry in the list must also have its own category defining the driver, device name, port, and type.

```
printers = { My Printer
             My PostScript to file }
```

9.2.24 [<printer device name>]

Each printer defined in the *printers* key of the *printer* category (see above) must have its own category. The name of the category must be the same as the printer named in the installed printers list of the *printer* category. The keys below define the printer's characteristics.

baudRate

```
baudRate = <number>
```

This key defines the printer's communication rate, for those printers which communicate via a serial connection.

device

```
device = <device name>
```

This key defines the device name of the installed print device.

```
device = Apple LaserWriter Plus v38.0 (PostScript)
```



driver

```
driver = <file name>
```

This key defines the print driver used for the installed print device.

```
driver = PostScript driver
```

handshake

9.2

```
handshake = <hardware, software>
```

This key indicates the type of handshake used by the printer for those printers which communicate via a serial connection. It must be one of the values specified above.

```
handshake = hardware
```

```
handshake = software
```

parity

```
parity = <none, even, odd, mark, space>
```

This key indicates the printer's parity type for those printers which communicate via a serial connection. It must be one of the values shown above.

```
parity = none
```

```
parity = even
```

port

```
port = <port name>
```

This key specifies the port to which the device is attached. This must be one of the values shown in the following examples.

```
port = LPT1
```

```
port = LPT2
```

```
port = LPT3
```

```
port = COM1
```

```
port = COM2
```

```
port = COM3
```

```
port = COM4
```



```
port = FILE
```

stopBits

```
stopBits = <number>
```

This key specifies the number of stop bits used by the printer for those printers which communicate via a serial connection. This should be 1, 1.5, or 2.

9.2

```
stopBits = 1
stopBits = 1.5
stopBits = 2
```

type

```
type = <number>
```

This key defines the type of device this installed print device is. The number indicates the PrinterDriverType as defined in **spool.goh**.

```
type = 0      ; PDT_PRINTER
type = 3      ; PDT_CAMERA
```

wordLength

```
wordLength = <number>
```

This key indicates the communication word length for those printers which communicate via a serial connection. This should be a number between five and eight inclusive.

```
wordLength = 8
```

9.2.25 [screen 0]

The *screen N* category is used to define the characteristics of each screen in the system. Currently, GEOS completely supports only one screen, which is called screen 0. This screen's characteristics are defined by the *screen 0* category.



device

device = <device name>

This key specifies the full device name of the screen's device. The standard devices are shown in the examples below.

```
device = Hercules HGC: 720x348 Mono
device = IBM MCGA: 640x480 Mono
device = CGA: 640x200 Mono
device = EGA: 640x350 16-color
device = VGA: 640x480 16-color
```

driver

driver = <file name>

This key specifies the file name of the driver used to run this screen.

```
driver = vga.geo
```

olddevice

olddevice = <device name>

When the user switches between video drivers, the system keeps track of the old device name in case the user made a mistake and wants to switch back. It will store the old value of the "device" field in this field.

olddriver

olddriver = <file name>

When the user switches between video drivers, the system keeps track of the old driver name in case the user made a mistake and wants to switch back. It will store the old value of the "driver" field in this field.



userdevice

```
userdevice = <device name>
```

GEOS doesn't use this field. The Debug utility uses this field to store the user's personal video device choice when simulating hardware devices that would not support the choice.

userdriver

9.2

```
userdriver = <file name>
```

GEOS doesn't use this field. The Debug utility uses this field to store the user's personal video driver choice when simulating hardware devices that would not support the choice.

9.2.26 [serial]

```
port <number> = <level>
```

The *serial* category defines all the serial ports available on the machine running GEOS. If a port is not defined in this category, GEOS will not recognize its existence. The single key in this category may be used numerous times, once for each available serial port.

The *port* key defines the hardware interrupt level for the specified port. The normal entries for the four base serial ports are shown in the examples below. If no value is specified, GEOS will try to generate an interrupt for the port and set the value itself. (It only checks levels three and four, though.) Note also that on an AT-class machine, level two for a card is actually level nine specified here.

```
port 1 = 4
port 2 = 3
port 3 = 4           ; may not work if port 1 is in use
port 4 = 3           ; may not work if port 2 is in use
```

9.2.27 [sound]

The sound category is used by the sound library to determine which sound driver is selected for the system running GEOS.

sampleDriver

```
sampleDriver = <driver file name>
```

9.2

This key specifies the sound driver that will be used to process all the sampled sounds produced by the system. If this key is not set, the standard sound driver (**standard.geo**) will be used.

```
sampleDriver = sbaster.geo
```

synthDriver

```
synthDriver = <driver file name>
```

This key specifies the sound driver that will be used for all synthesized sounds (beeps, UI sounds, etc.) If this key is not set, the standard sound driver (**standard.geo**) will be used.

```
synthDriver = standard.geo
```

9.2.28 [spool]

```
simpleUI = <Boolean>
```

The *spool* category has a single key used by the print spooler to configure its user interface. If true, the *simpleUI* key will display only a simple UI scheme; this is especially useful for small-screen devices because it significantly reduces the size of the print control dialog box. The default is false.

```
simpleUI = true  
simpleUI = false
```



9.2.29 [system]

The *system* category defines system configuration and setup. Most of the keys in this category are set and maintained by the Preferences manager application. These keys, with their formats and possible values, are shown in the following sections.

continueSetup

9.2

continueSetup = <Boolean>

If this key is set true, GEOS will begin by running the graphical setup program in the proper setup modes. If false, GEOS will bypass the graphical setup. After running, the graphical setup program will reset this field to false. This field overrides the *execOnStartup* key of the *ui* category.

```
continueSetup = true
continueSetup = false
```

drive

drive <letter> = <number>

This key allows you to override the drive-map initialization done by the primary filesystem driver. You can not make a driver believe a nonexistent drive exists, but you can change the presumed media or make the driver ignore a drive. More than one drive may be remapped.

The *letter* argument is the drive letter of the drive to be remapped. The *number* argument defines the new drive definition and is one of the following values:

-1	fixed disk
0	ignore the drive
360	360 K 5.25-inch disk
720	720 K 3.5-inch disk
1200	1.2 meg 5.25-inch disk
1440	1.44 meg 3.5-inch disk
2880	2.88 meg 3.5-inch disk

Some examples of drive remappings are shown below:

```
drive d = 0          ; ignore drive D:
drive a = 360        ; make GEOS think drive A: is 360K
```

font

font = <driver file names>

9.2

This key causes the named font driver to be loaded. If this key doesn't exist, **nimbus.geo** will automatically be loaded (the default driver). More than one driver may be specified on a single line or in a blob format, as shown in the examples below. (Note, though, that at current only **nimbus.geo** is recognized.)

```
font = nimbus.geo otherdrv.geo
font = { nimbus.geo
         otherdrv.geo }
```

fontid

fontid =

This key specifies the default font used in the event a requested font does not exist. This font will also be used when putting up system alert boxes (such as Abort/Retry boxes). The only available default font currently is Berkeley; typically, this will be a bitmap font rather than an outline font.

```
fontid = berkeley
```

fontmenu

fontmenu = <string of numbers>

This field specifies the order of fonts which should appear in font menus presented by font control objects. This is encoded as a string of numbers, four hex digits for each font, those four digits containing the font ID of the appropriate font. Thus, if the font ID's for the URW Roman and Berkeley fonts are 0x3000 and 0x0202, respectively, then if they are to be the first fonts in the font menu, the *fontmenu* field would read:

```
fontmenu = 30000202
```



fontsize

```
fontsize = <number>
```

This key specifies the point size of the default font. If an application requests a font that can't be found, the default point size specified here is used with the font specified with *fontid*. Berkeley supports 9, 10, 12, 14, and 18, though 18 is normally too large for many applications.

```
fontsize = 10
```

9.2

fonttool

```
fontmenu = <string of numbers>
```

This field specifies the order of fonts which should appear in font pop-up list presented by font control objects. This is encoded as a string of numbers, four hex digits for each font, those four digits containing the font ID of the appropriate font. Thus, if the font ID's for the URW Roman and Berkeley fonts are 0x3000 and 0x0202, respectively, then if they are to be the first fonts in the font pop-up list, the *fonttool* field would read:

```
fonttool = 30000202
```

fs

```
fs = <driver file names>
```

This key defines the file system drivers to be loaded. The kernel will by default attempt to load the primary IFS driver for the detected version of DOS; if it can not determine the primary IFS driver, the proper driver must be specified in the INI file under this key. Multiple file system drivers may be specified either on a single line or in blob format. The current secondary IFS drivers available are

netware.geo

Used for Novell Netware systems.

msnet.geo

Used for LANtastic and other networks that support the standard DOS device redirection calls.

cdrom.geo

Used for CD-ROM drives accessed through MSCDEX.EXE.



The INI File

406

```
fs = netware.geo
fs = { msnet.geo
      cdrom.geo }
```

handles

```
handles = <number>
```

9.2

This key specifies the number of handles GEOS should set as the maximum in the handle table. This should be set to something most likely 2000 or above, and it may be set in the Preferences manager application. If nothing is set in this key, the kernel will assume a default of 1000 handles.

```
handles = 2000
```

inkTimeout

```
inkTimeout = <number>
```

This key sets the number of ticks (60 ticks in a second) the system will wait after the user has stopped drawing before processing ink input. The default is nine tenths of a second, or 54.

```
inkTimeout = 54
```

maxTotalHeapSpace

```
maxTotalHeapSpace = <size of heap in paragraphs>
```

This field causes the **GeodeLoad()** routine to operate in transparent launch mode. The value given represents the overall size of the heap, in paragraphs, excepting system libraries that are always in memory. It should be determined on the target machine itself, by starting up, then running the TCL function “heapspace total”. A common value for this field is around 31000.

memory

```
memory = <driver file names>
```

This key defines the swap drivers that should be loaded. Swap drivers allow GEOS to use memory above the conventional 640 K. The kernel attempts to determine what type of memory is available and load the appropriate swap



driver. This key is settable by the user with the Preferences manager application. The four driver names recognized are

emm.geo LIM 4.0 standard expanded memory driver. A DOS-level memory driver must be loaded (e.g. EMM.SYS), typically in CONFIG.SYS.

extmem.geo 80286 extended memory driver.

xms.geo XMS/HIMEM.SYS driver. A DOS-level driver must also be loaded (e.g. HIMEM.SYS), typically in CONFIG.SYS.

9.2

disk.geo Disk swap driver. This should be loaded in all cases where a disk swap file is desired.

```
memory = disk.geo
memory = { disk.geo
           xms.geo }
```

noFontDriver

noFontDriver = <Boolean>

If true, this key tells GEOS not to load the font driver; this is useful only when it is known beforehand that outline fonts are not available; it will reduce startup time of GEOS. If the key does not exist, it defaults to false. If used improperly, this key can cause bad things to happen in the system.

```
noFontDriver = true
noFontDriver = false
```

notes

notes = <string>

This field isn't used by GEOS proper. The Debug utility will search for this field when looking for text describing a platform which the .ini file simulates.

noVidMem

noVidMem = <Boolean>

If true, this key tells GEOS not to load the vidmem driver, which is used for printing; it will reduce startup time of GEOS and should be used only if it is



known beforehand that printing will not be attempted. If the key does not exist, it defaults to false. If used improperly, this key can cause bad things to happen in the system.

```
noVidMem = true
noVidMem = false
```

pda

9.2

```
pda = <Boolean>
```

This field indicates whether GEOS is running on a PDA device. Currently this field is only used by the UI to provide alternate error strings.

penBased

```
penBased = <Boolean>
```

If true, this key tells GEOS that it is running on a pen-based system and that some objects will want to receive ink or other pen input. If the key is not set, it defaults to false.

```
penBased = true
penBased = false
```

power

```
power = <driver file name>
```

This key defines the power management drivers to be loaded, if any. If no driver is specified, the kernel will try to identify whether one is needed and then load it if necessary.

```
power = casio.geo
```

serialNumber

```
serialNumber = <number>
```

This key holds a predefined serial number for use by developers. This number will be given to you by Geoworks either directly or in the package you receive



containing GEOS. Normally, this number is entered by the user when GEOS first finishes its graphical setup program.

setupMode

```
setupMode = <number>
```

This key indicates the mode of the graphical setup program. This should be a number from zero to three; for full graphical setup, set it to zero. Other modes are internal in nature and should not be set.

9.2

```
setupMode = 0
```

splashColor

```
splashColor = <Color value>
```

If the *splashscreen* option has been turned on, this field will determine the background color of any text splash screens shown.

splashscreen

```
splashscreen = <Boolean>
```

This permits the GEOS loader to display a message on one of the five simple graphics mode screens while GEOS is loading.

splashText

```
splashText = <string>
```

If the *splashscreen* option has been turned on, this field will determine the text of the message to display.

9.2.30 [text]

The *text* category defines various characteristics of GEOS for the text objects, the spelling checker, and localization.

autoCheckSelections

autoCheckSelections = <Boolean>

If true, this key instructs the spelling checker to check the spelling of the selected text automatically when the user brings up the spell-check box. The default is true.

```
autoCheckSelections = true
autoCheckSelections = false
```

9.2

autoSuggest

autoSuggest = <Boolean>

If true, this key instructs the spelling checker to suggest other spellings automatically if a misspelling is found. The default is false.

```
autoSuggest = true
autoSuggest = false
```

dialect

dialect = <dialect code>

This key defines the dialect code used by the dictionary for spelling. Different dictionaries use different dialects within their own language. The Each dialect is represented by a number; the default setting is 128. The different dialects are listed below, by dictionary.

English	32	IZE British (realize/colour)
	64	ISE British (realise/colour)
	128	American (realize/color)
Dutch	64	Standard and non-preferred forms
	128	Standard Dutch forms only
French	64	Accents on uppercase characters
	128	No accents on uppercase characters
German	64	German Doppel s
	128	German Scharfes s
Norwegian	64	Nynorsk standard
	128	Bokmal standard



Portuguese	64	Brazilian Portuguese
	128	Iberian Portuguese

Some examples of setting the dialect are shown below.

```
dialect = 64
dialect = 128
```

dictionary

9.2

dictionary = <file name>

This key allows the user or the Preferences manager application to set the dictionary used by the spelling checker. The dictionary is set by specifying the file name of the dictionary data file; the default value is that for the English dictionary.

dictionary = IDNF9111.DAT	; Danish
dictionary = IENC9121.DAT	; English
dictionary = IFRF9121.DAT	; French
dictionary = IGRF9112.DAT	; German
dictionary = IITF9110.DAT	; Italian
dictionary = IPOF9110.DAT	; Portuguese
dictionary = ISPF9110.DAT	; Spanish

language

language = <number>

This key specifies the language in use by GEOS. The number is a language code (as shown in the examples below), and the user may set the language with the Preferences manager application. The default is English, 16.

language = 5	; French
language = 6	; German
language = 7	; Swedish
language = 8	; Spanish
language = 9	; Italian
language = 10	; Danish
language = 11	; Dutch
language = 12	; Portuguese



The INI File

412

```
language = 13           ; Norwegian
language = 14           ; Finnish
language = 15           ; Swiss
language = 16           ; English
```

languageName

```
languageName = <name of language>
```

9.2

This key specifies the name of the language in use; the default is American English. This key is normally set by the Preferences manager application.

```
languageName = American English
```

resetSkippedWordsWhenBoxCloses

```
resetSkippedWordsWhenBoxCloses = <Boolean>
```

If true, this key instructs the spelling checker to reset its list of skipped words when the user closes the spelling check box. The default is true.

```
resetSkippedWordsWhenBoxCloses = true
resetSkippedWordsWhenBoxCloses = false
```

smartQuotes

```
smartQuotes = <Boolean>
```

If true, this key instructs the text object to use “smart quotes,” quotation marks that curl themselves appropriately to their positions when typed. If this key is false, standard typewriter-style quotation marks will be used. The default is false; this is settable by the user in the Preferences manager application.

```
smartQuotes = true
smartQuotes = false
```



9.2.31 [ui]

autosave

autosave = <Boolean>

If true, this key tells GEOS to turn on the automatic backup feature; this may be set in the Preferences manager application.

```
autosave = true
autosave = false
```

9.2

autosaveTime

autosaveTime = <number>

This key indicates the number of seconds between autosave operations, if the *autosave* keyword is set to true. This may be set with the Preferences manager application.

```
autosaveTime = 300
```

background

background = <filename>

This key defines the file containing the picture to use as the background graphic. This is normally set by the Preferences manager application.

```
background = Bricks
```

backgroundattr

backgroundattr = <t, c, or x>

This key defines how the background picture should be displayed; it is normally set by the Preferences manager application.

- | | |
|----------|---|
| t | Tile the picture. |
| c | Center the picture on the screen. |
| x | Place picture in upper-left corner of the screen. |

```
backgroundattr = c
backgroundattr = t
```

backgroundcolor

```
backgroundcolor = <number>
```

This key defines the color of the background graphic. This is normally set by the Preferences manager application. The number is the color index of the color to be used.

```
backgroundcolor = 0
backgroundcolor = 12
```

deleteStateFilesAfterCrash

```
deleteStateFilesAfterCrash = <Boolean>
```

If true, this key tells GEOS to delete state files after every non-clean shutdown. If you set this, you will probably want to set the *doNotDisplayResetBox* key true as well.

```
deleteStateFilesAfterCrash = true
deleteStateFilesAfterCrash = false
```

doNotDisplayResetBox

```
doNotDisplayResetBox = <Boolean>
```

If true, this key tells GEOS not to display the system dialog box asking whether the user wants to reset the system or not after a crash. If you set this true, you should also set *deleteStateFilesAfterCrash* true, or some crashes may allow bad state files to keep GEOS from restarting properly.

```
doNotDisplayResetBox = true
doNotDisplayResetBox = false
```



execOnStartup

`execOnStartup = <program list>`

This key defines applications to be run when the UI is loaded (when GEOS starts up), named for their GEOS long names. The default is not to execute any additional programs.

```
execOnStartup = {           Lights Out Launcher
                           CD Player Application }
```

9.2

generic

`generic = <file name>`

This key defines the generic UI library that is to be used by GEOS. You will not need to set this; it will default to `ui.geo`.

```
generic = ui.geo
generic = uiec.geo
```

hardIconsLibrary

`hardIconsLibrary = <string>`

This is the long name of the library which provides the hard icon UI for a PC-based demo of a PDA device.

haveEnvironmentApp

`haveEnvironmentApp = <Boolean>`

If true, this key indicates that GEOS is using an environment application such as Welcome. If an environment application is being used, that application must be specified in the *defaultLauncher* key in the *uiFeatures* category. During debugging, you may set this key false and set the *execOnStartup* key to the application you're debugging to have GEOS come up directly into your application.

```
haveEnvironmentApp = true
haveEnvironmentApp = false
```



hwr

`hwr = <file name>`

This key indicates the handwriting recognition library to be loaded, if any. If GEOS is not on a pen-based system (*penBased* = *true* in the *system* category), then no handwriting recognition library will be loaded.

`hwr = palm.geo`

9.2

kbdAcceleratorMode

`kbdAcceleratorMode = <Boolean>`

If false, this key tells GEOS to ignore keyboard accelerators. By default, this is true and keyboard accelerators are allowed; this is independent of whether the accelerators are drawn or not. See also the *uiFeatures* category's *windowOptions* key.

`kbdAcceleratorMode = true`

`kbdAcceleratorMode = false`

noClipboard

`noClipboard = <Boolean>`

If true, this key prevents the UI from opening the clipboard file on startup. This is an optimization used when we want to open the clipboard only later on in a particular application.

noSpooler

`noSpooler = <Boolean>`

If true, this key prevents the UI from launching the spooler. This can be used to improve startup time if the system running GEOS knows beforehand that the spooler is not required. Very few systems will set this true.

`noSpooler = true`

`noSpooler = false`



noTaskSwitcher

noTaskSwitcher = <Boolean>

If true, this key prevents the UI from loading a task switch driver. This may be used to improve startup time if the system running GEOS knows in advance it will never use a task switcher.

noTaskSwitcher = true

noTaskSwitcher = false

9.2

noTokenDatabase

noTokenDatabase = <Boolean>

If true, this key prevents the token database from being initialized. This is useful as an optimization when GEOS will not need icons—that is, when GEOS starts up and runs just a single application.

noTokenDatabase = true

noTokenDatabase = false

overstrikeMode

overstrikeMode = <Boolean>

If false, this key prevents the user from switching into overstrike mode; it defaults to true, and it is settable in the Preferences manager application.

overstrikeMode = true

overstrikeMode = false

password

```
password = <Boolean>
```

This field turns the system password on or off.

passwordText

```
passwordText = <string>
```

9.2

Encrypted text of the password string, if any.

penInputDisplayType

```
penInputDisplayType = <number>
```

This key defines the PenInputDisplayType to be shown when the PenInputControl object is brought up. The PenInputControl object displays the floating keyboard in one of several display types. See the PenInputDisplayType enumerated type for definitions of its values.

```
penInputDisplayType = 1          ; floating keyboard  
penInputDisplayType = 7          ; handwriting area
```

productName

```
productName = <name>
```

This key holds the string displayed in the GEOS shutdown dialog box; for example, it will put up a string similar to “Are you sure you want to exit <productName>?”

```
productName = GEOS
```



screenBlanker

screenBlanker = <Boolean>

If this field true, then the user wishes to save the screen after an idle time period specified by means of the *screenBlankerTimeout* field.

screenBlankerTimeout

screenBlankerTimeout = <number of minutes>

9.2

If the user has turned on screen blanking, this is the number of minutes the system will stand idle before screen-saving turns on.

showTitleScreen

showTitleScreen = <Boolean>

If true, this key instructs GEOS to put up a title screen. This defaults to false.

```
showTitleScreen = true
showTitleScreen = false
```

sound

sound = <Boolean>

If true, this key instructs GEOS to turn sound on. If it's false, sound will be off. This is settable by the Preferences manager application.

```
sound = true
sound = false
```

specific

specific = <file name>

This key defines specific UI libraries to be loaded by GEOS. It defaults to *motif.geo*.

```
specific = motif.geo
```

tinyScreen

`tinyScreen = <Boolean>`

If true, this key tells GEOS that it's running on a small-screened device such as the Zoomer; it defaults to false. You can use this key during development if you're working on applications for a small-screen platform; it affects certain characteristics of the UI.

9.2

`tinyScreen = true`

`tinyScreen = false`

unbuildControllers

`unbuildControllers = <Boolean>`

If true, the UI will destroy the child blocks of controllers when the controller's menu/dialog box is closed. The child block will have to be regenerated every time the menu/dialog is opened—this is a memory for time tradeoff.

xScreenSize

`xScreenSize = <number>`

This key tells GEOS the screen width, in GEOS coordinates. If this key isn't set explicitly, the kernel will set it to the default screen size. This key is used primarily when developing for small-screen platforms such as Zoomer.

`xScreenSize = 256`

yScreenSize

`yScreenSize = <number>`

This key tells GEOS the screen height, in GEOS coordinates. If this key isn't set explicitly, the kernel will set it to the default screen size. This key is used primarily when developing for small-screen platforms such as Zoomer.

`yScreenSize = 344`



9.2.32 [<specific ui name>]

Each specific UI may have a category with options; this category should be named after the specific UI, e.g. [motif].

fontid

```
fontid = <font>
```

This field allows the user to specify a font that the specific UI should use when drawing text monikers for gadgets such as menus and buttons.

9.2

fontsize

```
fontsize = <size in points>
```

This field allows the user to specify a font size that the specific UI should use when drawing text monikers for gadgets such as menus and buttons.

9.2.33 [ui features]

The *ui features* category defines the UI configuration used by the environment application (e.g. Welcome) and all applications on the *execOnStartup* list in the *ui* category. On systems with no environment application, this category defines the UI configuration for all applications.

Related categories, *uiFeatures - intro*, *uiFeatures - beginner*, and *uiFeatures - advanced*, support the same keys. Each of these categories defines the configuration for a specific “room” of the Welcome application. Other environment applications will also use these keys for different “rooms.”

backupDir

```
backupDir = <relative path>
```

This key defines the directory in which the document control object will place quick-backup copies of document files. The default is PRIVDATA\BACKUP.

```
backupDir = DOCUMENT\BACKUP
```



defaultLauncher

defaultLauncher = <relative path>

This key defines the directory and application that acts as the default application launcher. This key should always have some application specified; otherwise, no application will start when GEOS loads. The path specified should be relative to the WORLD directory.

9.2

defaultLauncher = Utilities\GeoManager

docControlFSLevel

docControlFSLevel = <number>

This key specifies the document control's file selector user level. The file selector box has three different configurations; set the appropriate number (below) to determine which configuration is used.

0	No directories
1	Directories shown, simple UI configuration
2,3	Directories shown, complete UI configuration

An example of setting the file selector level is shown below.

docControlFSLevel = 2

docControlOptions

docControlOptions = <number>

This key turns on or off a number of other features in the document control object. The features are controlled by the bits set or clear in the number given. The five most significant bits of a 16-bit integer are used and have the following meanings, from most significant bit:

DCO_BYPASS_BIG_DIALOG

If set, this indicates that the big dialog box normally presented for New File/Open File/Use Template operations should be bypassed (not used). For advanced users, this bit should be clear. For novice users, this bit should be set.

DCO_TRANSPARENT_DOC

If set, this indicates that a "Switch Document" metaphor should be used in place of the New/Open/Close metaphor for



document management. This will allow only a single document open at a time and will immediately prompt if no document is open. For introductory and novice users, this bit should be set.

DCO_HAVE_FILE_OPEN

If set, this indicates that there is an Open button in the File menu (subject to specific UI rules). This is typically not set.

DCO_FS_CANNOT_CHANGE

If set, this indicates that the file selector used by the document control object can not change configuration; that is, the file selector will not offer the option of switching between full and simple configurations.

9.2

DCO_NAVIGATE_ABOVE_DOC

If set, this indicates that the document control object's file selector will allow the user to navigate directories. If cleared, the user may not navigate above the default document directory.

Some examples of usage of the docControlOptions key are shown below, with their translations into bit representation (five most-significant bits only are shown).

```
docControlOptions = 16384      ; Introductory
                    ; 16384 = 0x4000 = 01000...
docControlOptions = 0          ; Beginner
                    ; 0 = 0x0 = 00000...
docControlOptions = 4096       ; Advanced
                    ; 4096 = 0x1000 = 00010...
```

expressOptions

expressOptions = <number>

This key defines the configuration of the express menu. It sets and clears features based on the least significant 11 bits of a 16-bit number. Each bit, from the most significant used (bit 10) down to the least significant, is detailed below.

UIEO_GEOS_TASKS_LIST

If set, this indicates that the express menu should contain a list of currently-running applications.



UIEO_DESK_ACCESSORY_LIST

If set, this indicates that the express menu should contain a list of applications in the World\Desk Accessories directory.

UIEO_MAIN_APPS_LIST

If set, this indicates that the express menu should contain a list of applications in the World directory.

UIEO_OTHER_APPS_LIST

If set, this indicates that the express menu should contain a hierarchical list of applications in subdirectories of the World directory.

UIEO_CONTROL_PANEL

If set, this indicates that the express menu should contain a control panel area.

UIEO_DOS_TASKS_LIST

If set, this indicates that the express menu should contain a list of available DOS tasks accessible by a task switcher.

UIEO_UTILITIES_PANEL

If set, this indicates that the express menu should contain a utilities panel area.

UIEO_EXIT_TO_DOS

If set, this indicates that the express menu should contain an "Exit to DOS" type of trigger.

UIEO_POSITION

This is a three-bit field indicating where the express menu should appear. Three different values are allowed:

- 0 No express menu
- 1 In the top of the Primary window
- 2 In the lower left (just below the bottom of the screen)

Some examples of this key are shown below.

```
expressOptions = 617                      ; Introductory
; 617 = 0x0269 = 0000 0010 0110 1001
; The bits turned on are listed below:
; UIEO_DESK_ACCESSORY_LIST
; UIEO_CONTROL_PANEL
```



```

; UIEO_DOS_TASKS_LIST
; UIEO_EXIT_TO_DOS
; UIEO_POSITION = 1, upper left of window
expressOptions = 889          ; Beginner
; 889 = 0x0379 = 0000 0011 0111 1001
expressOptions = 2041         ; Advanced
; 2041 = 0x07F9 = 0000 0111 1111 1001

```

9.2

helpOptions

helpOptions = <number>

This key defines the configuration used by the help controller object. Specifically, it determines whether the help controller will automatically provide help triggers in the GenPrimary and in dialog boxes. The default is to allow help triggers to be created and displayed. Only the least significant bit of a 16-bit number is used, and that bit's significance is shown below.

UIHO_HIDE_HELP_BUTTONS

If set, this indicates that the help controller should not display help triggers in the GenPrimary or in dialog boxes.

```

helpOptions = 1          ; hide help triggers
helpOptions = 0          ; display help triggers

```

interfaceLevel

interfaceLevel = <number>

This key determines the interface level of applications that use the *ui features* category for their configurations. The four values allowed are shown in the examples below.

```

interfaceLevel = 0      ; Introductory
interfaceLevel = 1      ; Beginner
interfaceLevel = 2      ; Intermediate
interfaceLevel = 3      ; Advanced

```



interfaceOptions

```
interfaceOptions = <number>
```

This key determines two different features of the UI in general. It uses the two most significant bits of a 16-bit integer; the two bits have the following meanings.

UIIO_OPTIONS_MENU

If set, this indicates that an Options menu should exist.

UIIO_DISABLE_POPOUTS

If set, this indicates that the UI should not allow GIV_POPOUT GenInteraction objects to pop in an out.

```
interfaceOptions = 16384      ; No Options menu
interfactOptions = 32768      ; Popouts not allowed
```

launchLevel

```
launchLevel = <number>
```

This key controls the interface level of the applications allowed to be launched under the particular field ("room" of the environment application). It allows four values as shown in the examples below.

```
launchLevel = 0    ; Introductory
launchLevel = 1    ; Beginner
launchLevel = 2    ; Intermediate
launchLevel = 3    ; Advanced
```

launchModel

```
launchModel = <number>
```

This key controls how applications are started and exited. It allows four values, each of which defines a different level of user.

```
launchModel = 0    ; Transparent (user does not
                   ; realize he is starting an
                   ; application
launchModel = 1    ; Single instance only
launchModel = 2    ; Multiple instances allowed
launchModel = 3    ; Advanced features allowed
```



launchOptions

launchOptions = <number>

This key controls how applications are started and exited; specifically, it has a single flag which determines whether any applications are allowed to be in desk accessory mode. This defaults to true to allow desk accessories. The single flag is the most significant bit of a 16-bit integer.

UILO_DESK_ACCESSORIES

9.2

If set, this indicates that desk accessories should be allowed.

```
launchOptions = 32768    ; allow desk accessories
launchOptions = 0        ; do not allow them
```

quitOnClose

quitOnClose = <Boolean>

If true, this key forces the closure of all applications in a room before that room may be exited. This will cause state saving to be turned off. The default for this flag is false. One note: setting *quitOnClose* = true and *launchModel* = 0 can result in undesirable behavior.

```
quitOnClose = true
quitOnClose = false
```

windowOptions

windowOptions = <number>

This key controls different window system options. The precise interpretation of each flag is up to the specific UI. The high 8 bits form a mask of the bits to affect. The low 8 bits indicated whether the masked bits should be turned on or off. Of the eight bits, the high one is meaningless; the seven flags are listed below. You should not set these, however, unless you are familiar with the workings of the UI and the specific UI.

UIWO_MAXIMIZE_ON_STARTUP

If true, application primary windows will come up maximized. Desk accessory applications may override this behavior.

UIWO_COMBINE_HEADER_AND_MENU_IN_MAXIMIZED_WINDOWS

If true, the title bar and menu bar areas of maximized windows



will be combined to save screen space. Only the window gadgetry and menus are retained; title strings are eliminated.

UIWO_PRIMARY_MIN_MAX_RESTORE_CONTROLS

If true, window gadgetry for maximizing, minimizing, and restoring the window will be included on the screen.

UIWO_WINDOW_MENU

If true, a Window menu for keyboard control of minimize, maximize, restore, move, resize, and close operations will be provided. If false, only a “close” button will appear in the menu’s place.

UIWO_PINNABLE_MENUS

If true, menus will be pinnable.

UIWO_KBD_NAVIGATION

If true, keyboard accelerators and keyboard navigation will be enabled.

UIWO_POPOUT_MENU_BAR

If true, menu bars will be allowed to pop out to be dialog boxes. This should be used in limited situations because specific UIs may not provide gadgetry to restore the menu bar if the dialog is closed.

9.2.34 [welcome]

The *welcome* category defines configuration and usage characteristics of the Welcome environment application. Its keys may be useful to you during development, though you will probably not need them for your applications.

startupRoom

startupRoom = <name of room>

This key defines the room in which Welcome will start when GEOS is run. This is settable in the Preferences manager application; you will probably want to set this to the room most appropriate for your application to speed



startup when debugging. The default is no setting, which will cause Welcome to present its title screen.

```
startupRoom = 1           ; Beginner
startupRoom = 2           ; Intermediate
startupRoom = 3           ; Advanced
```

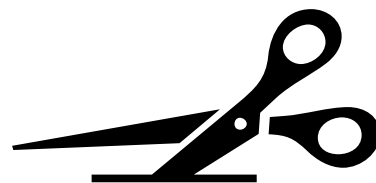
The INI File

430

9.2



Using Tools



10

10.1	Tools Summary	433
10.2	Typical Development Session	434
10.3	File Types	435
10.4	Esp	439
10.5	Glue	440
10.6	Goc	443
10.7	Grev	444
10.8	mkmf	447
10.9	pccom	448
10.9.1	PCCOM Background	448
10.9.2	Running PCCOM on the Target	449
10.9.2.1	Quitting PCCOM	449
10.9.2.2	Remote PCCOM Commands	450
10.9.2.3	Sending and Receiving Files	451
10.9.3	File Transfer Protocol of PCCOM	455
10.9.3.1	Sending a File to the Remote Machine	455
10.9.3.2	Retrieving a File Remotely	458
10.9.3.3	Calculating Checksum Values	459
10.10	pcget	461
10.11	pcs	462
10.12	pcsend	465
10.13	pmake	465
10.13.1	Copyright Notice and Acknowledgment	466
10.13.2	How to Customize pmake	467
10.13.3	Command Line Arguments	469
10.13.4	Contents of a Makefile	471
10.13.4.1	Dependency Lines	471



10.13.4.2	Shell Commands	474
10.13.4.3	Variables.....	476
10.13.4.4	Comments	480
10.13.4.5	Transformation Rules.....	481
10.13.4.6	Including Other Makefiles	486
10.13.4.7	Saving Commands	487
10.13.4.8	Target Attributes	487
10.13.4.9	Special Targets.....	488
10.13.4.10	Modifying Variable Expansion.....	490
10.13.5	Advanced pmake Techniques	493
10.13.5.1	Search Paths	493
10.13.5.2	Conditional Statements.....	495
10.13.6	The Way Things Work.....	497
10.14	Swat Stub	498



This chapter gives a reference of the tools included in this development kit. This chapter is not intended to teach you how to use the tools; it provides a reference only. To learn how to use most of the tools, see the Tutorial included in the development kit and the First Steps chapter in the Concepts book.

10.1

10.1 Tools Summary

The tools provided in this kit are listed below. Each is described in full in the following sections.

pmake	“Make” utility which has knowledge of Goc. The pmake tool manages geode compilation and linking, making appropriate calls to Goc and Glue.
Goc	C preprocessor which is aware of programming constructs specific to GEOS programming. The Goc tool looks for special keywords (such as @class and @object) and creates the proper GEOS data structures so that the geode will work correctly after being compiled and linked. Chances are you won’t call Goc directly but will instead use pmake , which will call Goc in turn.
Esp	Assembler with awareness of GEOS programming constructs. Also includes support for creating enumerated types, complicated records, and more. Chances are you won’t call Esp directly but will instead use pmake , which will call Esp in turn.
Glue	GEOS linker. This determines how the program’s resources should be set up and how those resources can be relocated. It determines file offsets and how to access library functions. As with Goc, you probably won’t call Glue directly but will instead use it through pmake .
grev	Revision number generator. This is a handy utility for generating revision numbers for geodes, which may be used to keep track of library protocols. The pmake program uses grev to keep track of incremental changes; if you make large changes to a geode at some point, you can use grev to modify your geode’s revision file to reflect your changes.
mkmf	Makefile maker. Creates a MAKEFILE which will in turn be used by pmake to determine how to create the geode’s files.



10.2	pccom	Communication manager. Sets up communication over serial lines. This manages data transfer and protocols when sending files between machines or debugging.
	pcs	Geode downloader. Sends geodes from the development machine to their proper directory on the target machine.
	pcsend	File downloader. Sends an arbitrary file from the development machine to an arbitrary directory on the target machine.
	pcget	File uploader. Retrieves an arbitrary file on the target machine, transferring it to any directory on the development machine.
	Swat	GEOS debugger. This program runs on the development machine, monitoring GEOS programs running on the target machine.

10.2 Typical Development Session

The Tutorial shows in detail how to write, compile, download, and debug an application. As a quick guide, however, the following list recaps the steps in a typical development session:

- 1 On the development machine, use a text editor (perhaps the one that came with your compiler) to write or edit the source (.c and .goc files, perhaps .asm or .ui files if you have the Esp assembler), header (.h and .goh files, perhaps .def files if you have the Esp assembler), and Glue parameters (.gp files) for your geode. The source files for the geode should be arranged within your development directory.
- 2 If you have never compiled the geode before or have added or included new files since the last compilation, you must run **mkmf** to create or update the MAKEFILE.
- 3 If you have added or included new files since the last compilation, you should execute **pmake depend** so that the compiler will know which files need to be remade if one is altered.
- 4 Run **pmake**. This compiles and links the geode's source files, using directions provided by the MAKEFILE.
- 5 Now you must download the compiled geode to the target machine. Use **pcs** to send the geode to the proper directory on the target machine (or **pcsend** if you have a particular destination directory in mind).



- 6 To test the geode, either run it on the target machine or run **swat** from the development machine to debug it.

10.3 File Types

You may be curious to know what sorts of files you'll be working with. If you have to work with someone else's code, then being able to find your way around their files (knowing which are sources, which are objects, and which are chaff) can be very useful.

10.3

- *.goc** These are Goc source files. You will write these files. They contain both standard C code and GEOS constructs (such as objects and messages).
- *.goh** These are Goc header files. You will write these files and include others. They provide definitions used by your .goc files (in the same relation as between standard C source and header files). Unlike standard C header files, these are included using the @include Goc keyword.
- *.poh, *.ph** These are files generated to optimize the @inclusion of .goh files. Goc will automatically generate these when they don't already exist.
- *.doh, *.dh** These are files generated to optimize the @inclusion of .goh files. They contain dependency information.
- *.c** These are standard C source files. You may write these as source files, using only standard ANSI C constructions. The Goc preprocessor will create .c files from .goc files. Thus, if you see two files with the same prefix, but one has the .c suffix and the other has the .goc suffix, then you know that the first was created from the second.

Note that Goc will create the generated .c file in the directory where it is invoked. Thus if your development tree contains files PROG\DIR1\CODE.GOC and PROG\DIR2\CODE.GOC, then if you convert these using Goc from the PROG directory, then one of the generated .c files will overwrite the other. Thus, you should never give .goc files the same prefix, even if they are in different directories.

- *.h** These are standard C header files. You may write these and include them using the ANSI C `#include` directive.
- *.asm** These are standard GEOS assembly source files, which may be assembled with the Esp assembler, if you have it. They may contain both standard assembly and Esp constructs.
- *.def** These are standard GEOS assembly header files, which you may write or include if you have access to the Esp assembler.
- *.mk, makefile** These are “makefiles,” files which contain scripts which the **pmake** tool will interpret and use to automatically compile and link your geode. In a source directory there will be a file called **MAKEFILE** (created with the **mkmf** tool) and probably a file called **DEPENDS.MK** (created by calling **pmake depend**). If you wish to customize how your geode is made, you will probably write a file called **LOCAL.MK**, containing your custom makefile script. The **INCLUDE** directory contains several .mk files, which will be `#included` by other makefiles.
- *.gp** These are “Glue parameter” or “geode parameter” files, which will give the Glue linker information necessary when linking a geode. You will write this file. The **pmake** program assumes that a geode’s .gp file will have name *geode*.GP, where *geode* is taken from the name of the directory containing the geode’s source (e.g. in the example above, **pmake** would expect the .gp file to be named **PROG.GP**).
- *.ldf** These are library definition files. Glue uses these files when linking your geode; they determine how your calls to a GEOS library will be encoded. If you are writing a library, then you will create one of these files by means of a **pmake lib**. The **pmake** program looks for .ldf files in the **INCLUDE\LDF** directory.
- *.rev** This is a revision file, used to keep track of a program’s revision and protocol levels (useful for tracking compatibility). The **pmake** tool will look for a file with name *geode*.REV, where *geode* is taken from the name of the directory containing the geode’s source (e.g. in the example above, **pmake** would expect the .rev file to be named **PROG.REV**). The **pmake** program uses **grev** to create and maintain the .rev file; you should use **grev** yourself when you need to signal a major revision.



*.rsc	Localization resource file. This file contains information which will be used by the ResEdit localization tool.	
*.obj	These are object files. These are files created by a C compiler or Esp which may be linked to form an executable. The pmake program uses Glue to link the object files.	
*.ebj	These are error-checking object files. GEOS supports the notion of “error-checking code.” When you write your programs, you can mark some commands as “error checking commands.” These commands might make sure that a routine is passed valid arguments or perhaps purposefully destroy some information which was not guaranteed preserved by a routine. Such commands may prove time-consuming but are useful for making sure that an application is robust. The pmake program will create two versions of your application—one which includes the EC (Error Checking) code, and one which doesn’t. Run the EC program to check for correctness, but use the non-EC version when the program should be fast (i.e. this is the version you should give to your customers). The .obj files will be linked to form a non-EC executable; .ebj files to form an EC executable.	10.3
*.geo	This is a Geode, a GEOS executable (either an application, library, or driver), the end result of your efforts. These are the files containing the code for GEOS programs which the user will interact with. They are created by linking together a number of .obj files, with additional information provided by a .gp file. You will place these files in your GEOS testing directory on your target machine (along with *ec.geo files, described below).	
*ec.geo	This is an error checking geode. (See above for quick descriptions of error checking code and geodes.) They are created by linking together a number of .obj and .ebj files, with additional information provided by a .gp file.	
*.sym	This is a symbol file, containing symbolic debugging information which the Swat debugger can use to access the geode’s data structures.	
*ec.sym	This is the symbol file of the error checking version of a geode.	
tmp*.*	These are temporary files which pmake will create and destroy while making your executable. The pmake program uses these	

files to pass arguments to the other tools. Thus, if you see a file of this name in your directory and you didn't create it, you can assume that **pmake** was interrupted in a recent make and was unable to erase the file (and thus it is safe for you to erase it).

.tcl, .tlc These are Tcl files, files containing Tool Command Language source code, used by the Swat debugging tool. The **.tcl** files contain Tcl source code, the **.tlc** files contain compiled Tcl code. The source code may be edited by any text editor and Swat will interpret it; compiled code runs more quickly, but can only be changed by editing the source code and re-compiling.

10.3

If you are writing a GEOS C application, you will write the following types of files:

- ◆ Source files: .goc files, optional .c files.
- ◆ Glue parameters file: *geode.GP*.
- ◆ Optional header files: .goh files, .h files.
- ◆ Optional custom make file: LOCAL.MK.

After you have made your geode the first time (creating a makefile with **mkmf**, a dependencies file with **pmake depend**, and the geode itself with **pmake**), your directory should contain the following additional file types:

- ◆ Intermediary C files: .c files (made by transforming .goc files).
- ◆ Makefile: MAKEFILE
- ◆ Dependencies file: DEPEND.MK
- ◆ Revision file: *geode.REV*.
- ◆ Object files: .obj and .obj files.
- ◆ Symbol files: .sym files
- ◆ Geodes: .geo files.



10.4 Esp

Esp (pronounced “esp”) is the GEOS assembler. It creates object files from Esp code—said code using a superset of MASM syntax. These object files may then be linked by means of the Glue tool.

Most users will never call Esp directly, instead going through **pmake**, which will make the proper calls to Esp for the most common cases.

10.4

Esp takes the following options:

-2 Code produced should be DBCS—characters will be two bytes instead of one.

-I *directory* Specifies an additional directory in which to look for included files.

-o *filename* Name to give to created object file.

-w *warntype*

-W *warntype* Turn warnings off or on

unref	Warn if a symbol that can only be used during this assembly isn't.
field	Warn if structure field used with . operator when lhs isn't of the type that contains the field
shadow	Warn if a local variable or procedure overrides a definition in a larger scope
private	Warn if a private method or instance variable is used outside a method handler or friend function related to the class that defined the thing
unreach	Warn about code that cannot be reached. This is very simplistic unknown Warn if a far call is made to a routine whose segment is unknown
record	Warn if a record initializer doesn't contain all the fields of the record
fall_thru	Warn if a function falls into another one without a .fall_thru directive
inline_data	Warn if a variable is defined where execution can reach

	unref_local	Warn if a local label isn't ever referenced
	jmp	Warn if out-of-range jumps are transformed into short jumps around near jumps
	assume	Warn when override is generated based on segment assumptions
	all	Used to turn all warnings on or off.
10.5	-M	This assembly is just for the purpose of determining what the source file's dependencies is. Instead of creating an object file, Esp will create a temporary file which the dependencies maker will use to determine the dependencies.
	-d	Activate's Esp debugging mode. Useful only when trying to track down a bug in Esp.

10.5 Glue

Glue is the GEOS linker. It creates GEOS or DOS executables from object files. (It can also create GEOS VM and font files, if you have the appropriate tools.) These object files may have been created by a C compiler, or by the Esp assembler. To create the executables, Glue must create a combined file, resolve external declarations, determine how to call libraries, and apportion the code and data into resources. Glue will also create a .sym file—a file containing symbolic debugging information which the Swat debugger will use for viewing the geode's data.

Most users will never call Glue directly, instead going through **pmake**, which will make the proper calls to Glue for the most common cases.

The Glue application takes the following arguments:

```
glue @file
glue <flags> <objFile>+ [-l<objFile>]*
```

@file

The Glue linker should take its arguments from the file *file* in addition to those on the command line. This may come in handy if you often use the same long string of options. Since you may need to pass Glue more arguments than may be input on the command line, sometimes this option is necessary.



Note that if you use this option, then all arguments must be included in the file—there should be no others on the command line itself.

The **pmake** program uses this option to pass arguments to Glue.

- L *path*** Specifies where Glue looks for .ldf (library definition) files. These files are placed in a standard directory by the system makefiles on a “**pmake lib**”.
- N *string*** Specify a copyright notice.
- Oe** Creating DOS executable (“*.exe*”) file. Of course, this option is not valid if the object files contain GEOS directives which will not work outside GEOS. 10.5
- Oc** Creating DOS command (“*.com*”) file. Of course, this option is not valid if the object files contains GEOS directives which will not work outside of GEOS.
- Og *file*** Creating GEOS executable (“*.geo*”) file. You must provide the name of the geode’s Glue parameters file (the *.GP* file). For information about setting up a parameter file, see “First Steps: Hello World,” Chapter 4 of the Concepts Book.

When creating a *.geo* file, you may pass any of the following options:

- E** Link Error checking version of geode.
- R *number*** Specify release number of geode (e.g. 3.2.1.0).
- P *number*** Specify protocol number of geode (e.g. 1.0).
- T *number*** File type.
- l** Creating a library; Glue should create .ldf file.
- Ov** Creating GEOS Virtual Memory (*.vm*) file. Using this option, you may create *.vm* storage files as set up by the Esp assembler. You may pass the following options when creating *.vm* files (if you don’t know the meaning of some of these terms, see “Virtual Memory,” Chapter 18 of the Concepts Book):
 - A *number*** **VMAttributes** to use for the file.
 - C *number*** Compaction threshold.
 - M *string*** Map block segment name.
 - P *number*** Protocol number (e.g. 2.5.0.3).

Using Tools

442

10.5

- R *number*** Release number (e.g. 12.3).
- i *name*** A .geo file from which to get the table of imported libraries. This allows the VM file to be opened by that geode and objects in the file to be used.
- t *token*** File token
- c *token*** Creator's manufacturer token.
- l *string*** File's long name.
- u *notes*** File's user notes.
- N *string*** Copyright notice which will be embedded in header of .GEO or .VM files.
- G *number*** You should never have to use this option. This specifies a non-standard GEOS release number (e.g. 1.3); if this release used a different VM header than GEOS 2.0, Glue will still construct the proper header as long as this option is passed. Since the default release number is 2, developers for 2.X do not need this option.
- Wall** Requests that Glue output all optional warnings.
- Wunref** Requests that Glue output optional unreferenced global symbol warnings.
- d** Dump memory. Normally used only when debugging Glue.
- m** Provide memory map in output. This information gives information about the sizes of various parts of the geode. This information proves especially helpful when making geodes work with small devices.
- nll** Disables the output of line numbers for local memory segments for any application with 163 resources or more.
- o *file*** Specify name of output file (e.g. WOROPRO.GEO).
- q** Leave the symbolic information behind even if an error was encountered. Normally, this flag is used only when debugging Glue.
- r** Maps segment relocations to non-shared resources to resource IDs. This is normally used only by multi-launchable C applications. When running more than one instance of a multi-launchable application, the system only uses one copy of the read-only portion of the application. The system makes separate copies of the writable data for each instance of the application. This can lead to conflict when the relocation instructions for the read-only data uses the handle or segment of a writable resource;



which copy should be responsible for providing these addresses? If you don't pass this flag and Glue detects the above situation, Glue will simply refuse to make the application multi-launchable. This flag instructs Glue to instead use the resource ID of the writable resource where it would normally use the segment or handle of that resource.

If you use this option, make sure that if you use the address of a variable in a resource other than dgroup that you use **GeodeGetOpPtrNS()** and lock down or dereference the handle of the returned optr.

- s file** Specify name of symbol file (the .SYM file).
- z** Output localization information.

10.6

10.6 Goc

Goc is the GEOS C preprocessor, which will turn your .goc file into something a regular C compiler can understand. It will traverse the .goc file, detect Goc keywords (e.g. @class, @object), turn these keywords into appropriate pieces of code, and write the resulting file out to a .c file. Note that Goc acts as a simple filter, and will only make changes where it detects Goc constructs; it won't touch your regular C code at all.

Under normal circumstances, you will not invoke Goc directly. Instead, **pmake** will make calls to Goc when compiling .goc files.

If for some reason you do need to invoke Goc directly, you may wish to know about its command line arguments:

```
goc @file
goc [args] <file>
```

- @file** Command-line arguments are stored in a file—Goc will read this file and treat the contents as its arguments. If you use this option, then all the arguments must be in the file—no others may appear on the command line itself.
- Redirect output to standard out.
- Dmacro** Set variable macro to one (e.g. “-D__GEOS__”)
- Dmacro=value** Set variable macro to value.

10.7

-I <i>dir</i>	Specify additional include directory
-I-	Turn off relative inclusion.
-L <i>name</i>	Specify name of library being compiled. Must match argument to @deflib Goc keyword in library's .goh file, if it has one.
-M	Help pmake generate dependency information.
-cb	Generate information for Borland compiler.
-d	Turn on all debugging information.
-dd	Output @default debugging information. (See "GEOS Programming," Chapter 5 of the Concepts Book to find out what @default means).
-dl	Output lexical analyzer debugging information.
-dm	Output message parameter definition debugging information.
-ds	Output symbolic debugging information.
-du	Output Goc-specific debugging information.
-dy	Output parser debugging information.
-l	Do localization work.
-o <i>file</i>	Specify name of output file.
-p	For Geoworks use only.

10.7 Grev

GEOS supports two version numbers for each geode. The first of these is the release number, used to uniquely identify the release of the geode. The protocol number tracks the external interface of the file. This is used to determine what versions of related geodes can be used together. The kernel will use these numbers to prevent loading of incompatible executable files.

The grev utility generates proper revision numbers. Normally, it is called automatically by **pmake**, so if you are just making a small change to a file, you need not call it directly. However, you may wish to. When using grev, you must think about how major a change you are making; a large change means



that you should change an earlier number of the release number. A change from 1.2.3.4 to 2.0.0.0 should signal a larger step than a change to 1.2.3.5.

The **pmake** program uses grev to automatically create revision numbers for geodes; it passes these values to Glue, which in turn places the protocol and revision numbers in the .geo and .sym files.

There are three widely used methods for incrementing release numbers with respect to public releases (for which a specific release number is desired for marketing, say “2.0.0”). The problem comes because it is not known until after a release has been built whether it will be *the* release or not (since bugs may be found).

10.7

The first method is to keep separate public release numbers and internal release numbers. This is awkward and confusing and is generally done when it is too late to do anything else.

The second method is to number successive revisions “1.14.0.12”, “1.14.0.13”, “1.14.0.14” and so on until the final revision is made which is numbered “2.0.0.0”. The problem with this is that one never quite knows whether or not a revision is the final one (since bugs may be found).

The third method is to number successive revisions “2.0.0.12”, “2.0.0.13”, “2.0.0.14” and so on. The released “2.0.0” revision is then the last engineering revision starting with “2.0.0.X”. The disadvantage of this method is that it can seem non-obvious at first and requires a little bookkeeping to know the engineering number of the released version.

The protocol number is changed whenever the external interface for the file changes. For the kernel and for libraries the protocol reflects the order as well as the parameters and behavior of external entry points. For applications the protocol reflects the object names, types and attributes. Changes that do not affect the external interface (changing the implementation of a routine, changing the moniker or hints of an object) do not change the protocol number.

The major protocol number reflects incompatible changes in interface, such as rearranging the order of entry points. The minor protocol number reflects upwardly compatible changes in the protocol (such as adding an entry point at the end of a jump table or using a bit formerly marked as “reserved”).

Each executable file contains protocol compatibility information (a protocol number) for all other executable files on which it depends. For example, a simple application might be compatible with kernel protocol “34.2” and UI protocol “19.7”. Thus the application is compatible with kernels “34.2” through “34.65535” and with UIs “19.7” through “19.65535”.

A protocol number is also stored with each state file to determine if the state can be recovered by the currently running application.

10.7

The `grev` tool uses a file (normally marked with a `.rev` suffix) in the geode's development directory to keep track of the revision number. The file is organized chronologically, with later entries at the beginning of the file. It contains

- ◆ one line for each compilation, denoting the revision number (which is incremented on each compilation), optional user name, the date, and an optional comment. By default, **pmake** will only change the last part of the release number.
- ◆ one line for each protocol change, denoting the protocol number, optional user name, the date, and an optional comment.

The `grev` utility takes the following arguments:

new file [*comment*] [-P] [-R]

Create a new revision record, listing *comment* as an initial revision for the base (0.0.0.0 release, 0.0 protocol). This command may only be executed in the geode's development directory. The `-P` option causes `grev` to give minimal output, printing only the protocol number. The `-R` option causes `grev` to print only the revision number. These last two options are normally used by **pmake** to extract the relevant numbers.

info file Print the current release and protocol from the revision file.

getproto file

Print only the current protocol from the revision file.

`newprotomajor file` [*comment*] [-P] [-R]

NPM file [*comment*] [-P] [-R]

Increase the major protocol number by one, setting the minor number to zero. The *comment* argument is listed as the reason for the change in the file. The `-P` and `-R` options work as they do for **grev new**.

`newprotominor file` [*comment*] [-P] [-R]



npm file [*“comment”* | -P | -R]

Increase the minor protocol number by one. The comment string is listed as the reason for the change in the file. The -P and -R options work as they do for **grex new**.

newrev file number1.number2 [*“comment”* | -P | -R]

Increase release number from A.B.C.D to *number1.number2*.0.0. The comment is listed as the reason for the change. The -P and -R options work as they do for **grex new**.

newchange file [*“comment”* | -P | -R]

10.8

nc file [*“comment”* | -P | -R]

Up release number from A.B.C.D to A.B.C+1.0. The comment is listed as the reason for the change. The -P and -R options work as they do for **grex new**.

neweng file [*“comment”* | -P | -R]**ne file** [*“comment”* | -P | -R]

Increase release number from A.B.C.D to A.B.C.D+1. The comment is listed as the reason for the change. The -P and -R options work as they do for **grex new**.

10.8 mkmf

The mkmf tool exists to create a file named MAKEFILE. The **pmake** program will use this file as a sort of script, using it to determine how to compile and link the geode. However, makefiles can get rather complicated, so it is best to create them using mkmf instead of by hand.

For information about customizing this boilerplate makefile, see section 10.13 on page 465.

The mkmf tool uses the following rules to build the makefile:

- ◆ The geode name is taken from the name of the directory in which you run mkmf. Among other things, this means that if you change the name of your development directory, you should also run mkmf again.
- ◆ Any file in the current directory or its subdirectories will be considered a source file if it has one of the following suffixes: .asm, .def, .ui, .c, .h, .goc,

or .goh (except that .c files which are generated from .goc files will not be considered sources).

- ◆ For each .asm, .c, or .goc file, an .obj file with corresponding name will be added to the makefile variable OBJS.
- ◆ If there are subdirectories, then each subdirectory is considered to hold the source for a *module* of the program, and **mkmf** will work with the files in this subdirectory as a unit. The module's name will be added to the CMODULES variable if it contains .c or .goc files; if it contains .asm files, then it will be added to the MODULES list. When using **pmake**, each module will be considered something that can be made, a sort of intermediate step towards making the whole geode. If you do not wish the files in a subdirectory to be incorporated in the program, create a file in the directory called NO_MKMF. This file need have no contents.

10.9

10.9 pccom

The **pccom** tool manages communication between the development and target machines. It assumes that the machines are connected by a single serial line. All I/O is interrupt-driven with XON/XOFF flow control active on the development machine and obeyed on the target machine.

Note that it is possible to use some features of pccom from within GEOS. For more information about this, see “PCCom Library,” Chapter 22 of the Concepts Book.

10.9.1 PCCOM Background

PCCOM is used in two primary situations. First, it is used by GEOS software developers when transferring files or when debugging an application. In this situation, the developer runs PCCOM on the target machine and then runs PCS, PCSEND, or PCGET, on the host machine. All of these programs know how to interact with PCCOM.

Second, it is used by DOS programs when transferring files to and from Zoomer devices or other devices that require remote file manipulation. In this



case, the host machine runs a program which copies escape character sequences to the appropriate serial port, prompting PCCOM to act.

10.9.2 Running PCCOM on the Target

PCCOM is a DOS program that monitors the serial port and responds to commands received on the line. All I/O is interrupt driven with XON/XOFF flow control active on the host machine and obeyed on the target machine.

10.9

PCCOM uses the PTTY environment variable of DOS, if it exists. This variable contains communications settings detailing baud rate, COM port, and communications interrupt level. You can override the PTTY settings with command-line options to PCCOM when running it. The following command-line options are allowed:

- /b:baud** Specify the baud used for file transfer and serial communications. The baud parameter may be one of the following values: 300, 1200, 2400, 9600, 19200, or 38400. Unambiguous abbreviations may be used (e.g. 9 for 9600 baud or 38 for 38400 baud). The default baud rate is 19200 bps.
- /c:port** Specify the COM port used for serial communications. The parameter may be one of 1, 2, 3, or 4. The COM ports for Zoomer are
- | | |
|-------|---|
| COM 1 | the built-in serial port (this is the default). |
| COM 2 | the infrared transceiver. |
- /i:interrupt** Specify interrupts that should be ignored by PCCOM. This is useful if peripherals share an interrupt number that may confuse PCCOM. The interrupt parameter is one or more numbers of the interrupt(s) to be ignored, in hexadecimal.
- /l:irq** Specify the IRQ level for serial line communications. This parameter is rarely required. The irq parameter is the number of the IRQ level to be used.

10.9.2.1 Quitting PCCOM

PCCOM may be quit either directly or remotely. To quit PCCOM directly, simply hit the Enter key (or the q key) on the machine on which PCCOM is running. If it does not quit on the first keystroke, hit the key again.



To quit PCCOM remotely, issue the quit escape sequence <Esc>EX through the serial line from the host machine. See below for a description of the commands that can be issued remotely.

10.9.2.2 Remote PCCOM Commands

10.9

PCCOM doesn't care what machine originates a remote command; its sole purpose is to evaluate and execute commands received through the serial port it's monitoring. Thus, a command sent by one Zoomer to another will exact the same response as a command sent by a development host machine to a target development machine.

On most computers, commands are copied from DOS to the serial port using the "echo" command. (If the computer executing remote commands has a different BIOS, you may need to access the serial port differently; in this case, you must make sure that the characters sent to the serial port in the end are the same as those shown in the table below.) For example, to send the "quit" command to the remote machine, you could use the DOS command

```
C:>echo EscEX > com1
```

where *Esc* (in italics) represents the Escape character (0x1B).

No matter what method you use to send the character sequences to the serial port, the following commands may be executed remotely. Sending and receiving files remotely is more involved and is therefore discussed in the next section; it is not complicated, however.

All arguments to PCCOM remote commands must end with an exclamation point. Although this character is normally an acceptable character within DOS file names, PCCOM will treat it as the argument delineator character. Because of this, file operation commands will not work on files with exclamation points in their names.

Command	Sequence	Description
Send File	<Esc>XF1	Send a file from the host to the remote machine using the PCCOM file transfer protocol (see below).
Get File	<Esc>XF2	Retrieve a file from the remote machine using the PCCOM file transfer protocol (see below).
Copy File	<Esc>CPsrc!dest!	Copy the file named in the src argument to the file named in the dest argument. File name arguments may be full or



		relative paths with or without drive letters. This is equivalent to the DOS COPY command.	
Move File	<Esc>MV <i>src!dest!</i>	Move the file named in the <i>src</i> argument to the file named in the <i>dest</i> argument. File name arguments may be full or relative paths with or without drive letters. This is equivalent to the DOS MOVE command.	
Delete File	<Esc>RF <i>file!</i>	Remove the named file; the file argument may be a full path or a file in the current directory. This is equivalent to the DOS DEL command.	10.9
Change Drive	<Esc>CD <i>drive!</i>	Change the working volume to the drive named in the drive argument. This is equivalent to changing the drive in DOS by typing the drive letter followed by a colon (e.g. C:).	
Change Directory	<Esc>CD <i>dir!</i>	Change the working directory to that named. The <i>dir</i> argument may be a full or relative path; this is the equivalent of the DOS CD command.	
Show Current Path	<Esc>CD!	Print the current directory's path. This is equivalent to the DOS CD command with no arguments passed.	
List Files in Dir	<Esc>LS	List files in the current working directory. This is equivalent to the DOS DIR command with no arguments.	
Create Directory	<Esc>MD <i>dir!</i>	Create a new directory according to the <i>dir</i> argument. The <i>dir</i> argument may be a full or relative path. This is the equivalent of the DOS MKDIR command.	
Delete Directory	<Esc>RD <i>dir!</i>	Remove the directory named in the <i>dir</i> argument. The <i>dir</i> argument may be a full or relative path; this is equivalent to the DOS RMDIR and RD commands.	
Clear Screen	<Esc>cl	Clear the screen. This is equivalent to the DOS CLS command.	
Exit PCCOM	<Esc>EX	Exit PCCOM on the remote machine.	

10.9.2.3 Sending and Receiving Files

If you are using the GEOS SDK, you will do most of your file sending and receiving using the programs PCS, PCSEND, and PCGET. These programs send commands to the serial port, and then follow them by either providing or receiving packaged file data. These three programs are detailed below; following them is a section of the file transfer protocol of PCCOM if you are writing your own remote-access program(s).



PCGET

If PCCOM is running on the target machine, the PCGET program can be executed on the host to retrieve a file from the target. This simple program merely retrieves the file and copies it into the host's working directory under the same name.

PCGET takes the following arguments; only the file name is required. The other arguments are optional and may be used to override the settings in the host machine's PTTY environment variable (see above, under "Running PCCOM on the Target (or on the Zoomer)").

```
pcget [/b:baud][ /c:port][ /I:irq] file
```

- /b:baud** Specify the baud used for file transfer. The baud parameter may be one of the following values: 300, 1200, 2400, 9600, 19200, or 38400. Unambiguous abbreviations may be used (e.g. 9 for 9600 baud or 38 for 38400 baud). The default baud rate is 19200 bps.
- /c:port** Specify the COM port used. The port parameter may be one of 1, 2, 3, or 4. The COM ports for Zoomer are
COM 1 the built-in serial port (this is the default),
COM 2 the infrared transceiver.
- /I:irq** Specify the IRQ level for the transfer. This parameter is rarely required. The irq parameter is the number of the IRQ level to be used.
- file** Specify the file to be retrieved. The file parameter may be a full or a relative path or a simple file name. The file will be copied from the target to the host into the host's current working directory, with the same name.

PCSEND

If PCCOM is running on the target (remote) machine, PCSEND may be executed on the host machine to download a file to the target. PCSEND will only send a single file, though it may send the file to any directory on the target. To send multiple files, or to download specific geodes to their proper locations in the GEOS 2.0 directory tree, use the PCS program instead.



The command line options of PCSEND are shown below. Only the file to be sent is required; if no other argument is passed, the file will be sent to the target's current working directory.

```
pcsend [/b:baud][/c:port][/I:irq] file [/d:dest]
```

/b:baud	Specify the baud used for file transfer. The baud parameter may be one of the following values: 300, 1200, 2400, 9600, 19200, or 38400. Unambiguous abbreviations may be used (e.g. 9 for 9600 baud or 38 for 38400 baud). The default baud rate is 19200 bps.	10.9
/c:port	Specify the COM port used. The port parameter may be one of 1, 2, 3, or 4. The COM ports for Zoomer are COM 1 the built-in serial port (this is the default), COM 2 the infrared transceiver.	
/I:irq	Specify the IRQ level for the transfer. This parameter is rarely required. The irq parameter is the number of the IRQ level to be used.	
file	Specify the file to be sent. The file parameter may be a full or a relative path or a simple file name. The file will be downloaded to the target machine's current working directory, unless the /d parameter is also passed (see below).	
/d:dest	Specify a full or relative destination path for the file and/or a new destination file name.	

PCS

If PCCOM is running on the target machine, PCS may be executed on the host machine to send multiple files to predetermined directories on the target. PCS is most often used by GEOS developers using the GEOS development kit, when they are downloading their recently-compiled geodes to the target for debugging.

The PCS program makes use of a list of constraints-tokens and their source and destination files and directories-located in the files
ROOT_DIR\INCLUDE\PCS.PAT and ROOT_DIR\INCLUDE\SEND on the SDK
host machine. (ROOT_DIR is a DOS environment variable set up by the SDK
installation program indicating the top directory into which the SDK files
were installed.) The format of these two files is described at the end of this
section.



The command-line parameters of PCS are shown below. Note that a file name is not used by PCS; instead, if no token or directory is given, PCS will download all appropriate files in the current working directory. As with PCSEND and PCGET, the baud, COM port, and IRQ level arguments are all optional and may be used to override the settings in the PTTY environment variable.

```
pcs [/n][/Sf][/t][/b:b][/c:p][/I:i][dir|file|token]
```

10.9

- /n** If /n is specified, PCS will send non-EC geodes only. Without this argument, PCS will send only EC geodes.
- /Sf** Specify a file containing a list of files to be sent. The file argument is the name of the file.
- /t** If the /t argument is used anywhere on the command line, file names specified at the end of the command (see the last argument) will be interpreted as tokens. A token may equate to numerous files as defined in the SEND file.
- /b:b** Specify the baud used for file transfer. The baud parameter may be one of the following values: 300, 1200, 2400, 9600, 19200, or 38400. Unambiguous abbreviations may be used (e.g. 9 for 9600 baud or 38 for 38400 baud). The default baud rate is 19200 bps.
- /c:p** Specify the COM port used. The port parameter may be one of 1, 2, 3, or 4. The COM ports for Zoomer are
COM 1 the built-in serial port (this is the default),
COM 2 the infrared transceiver.
- /I:i** Specify the IRQ level for the transfer. This parameter is rarely required. The irq parameter is the number of the IRQ level to be used.

dir|file|token

Specify a directory containing the geodes to be downloaded, the files to be downloaded, or tokens to be interpreted. If no directory, file, or token is specified, PCS will download the appropriate files in the current working directory. If a directory is specified, PCS will download all the appropriate files in that directory. If file names are specified (multiple files and/or directories may be specified), all affected files will be send.



If the /t argument appears anywhere in the command line (see above), this set of arguments will be interpreted as tokens. See directly below for token use and interpretation.

When using a token, PCS looks in the ROOT_DIR\INCLUDE\SEND file for the token to determine which files should be sent. Generally, all executables associated with an application, library, or mechanism are sent when the appropriate token is passed. Look in the SEND file to find out what the accepted tokens are and what they send.

10.9

For example, if the SEND file contained the lines

```
PC      DRIVER/VIDEO/DUMB/HGC/HGC      GEO
PCB     DRIVER/VIDEO/DUMB/HGC/HGC      GEO
PCB     DRIVER/MOUSE/LOGIBUS/LOGIBUS    GEO
```

then typing

```
pcs /t pc
```

would send just the HGCEC.GEO file to the proper directory on the target.
Typing

```
pcs /t pcb
```

would download both HGCEC.GEO and LOGIBUSE.GEO to their proper directories. A listing of all the supported tokens can be found in the SEND file.

10.9.3 File Transfer Protocol of PCCOM

If you need to create your own file transfer program or module, you can use the basic PCCOM commands and a special transfer protocol to send or receive files over the serial link. This is useful, for example, if you have an existing Windows or DOS program to which you would like to add the ability to transfer files to or from the Zoomer (or another unit running PCCOM).

10.9.3.1 Sending a File to the Remote Machine

Sending a file to the remote machine involves the steps below. A file may be sent by any program that can access the serial port.



- 1** Notify PCCOM that a file is on its way.
Send the Send File escape character sequence to the serial port, notifying PCCOM that a file is about to be sent to it. The escape sequence is <Esc>XF1.
- 2** Send the destination file name.
Send the name PCCOM should use for the file when saving it. The name is a string of sequential characters ending with a null byte. If sending the file to the target machine's target directory, this will just be the file's name; if sending to a different directory, this string should instead be the full pathname. Thus, to do the equivalent of "pcsend yuyuhack.sho /d:b:\geoworks\document", this string would consist of "b:\geoworks\document\yuyuhack.sho".
- 3** Wait for acknowledgment of the name transfer.
PCCOM will send a 0xFF character or a SYNC byte to acknowledge acceptance of the file name. If you do not receive this character, an error has likely occurred.
- 4** Send the file size.
The file size should be encoded as a dword value. Send the low byte first.
- 5** Send a packet.
Once you have received the SYNC or 0xFF byte, you can safely begin sending packets of data to PCCOM. A packet has the following format (sequence of bytes, with the first listed being the first sent):

```
BLOCK_START          ( = 1 )  
data  
BLOCK_END            ( = 2 )  
CRC                  ( checksum value )
```

The data between BLOCK_START and the checksum value (CRC) may be up to 1 K. In order to avoid PCCOM confusion between a normal data byte and a BLOCK_START or BLOCK_END, a third element—BLOCK_QUOTE—is used.

Any time you have a data byte equal to 1, 2, or 3, you must quote it by inserting a BLOCK_QUOTE byte before it and then adding three to its value. Thus, if you had a data sequence consisting of the following

```
100, 42, 2, 3, 16
```

you would send the following sequence of bytes to transfer the data:



```

<Esc>FX1                                ( alert PCCOM a
                                         file is coming )

<null-terminated file name>
<wait for SYNC byte>
BLOCK_START                              ( = 1 )
100
42
BLOCK_QUOTE                              ( = 3 )
5                                         ( = 2 + 3 )
BLOCK_QUOTE                              ( = 3 )
6                                         ( = 3 + 3 )
16
BLOCK_END                                ( = 2 )
CRC                                      ( checksum value )

```

10.9

The CRC word is two bytes of checksum value as calculated using the table and code shown in “Calculating Checksum Values,” below. The CRC value is based upon the data bytes only. The low byte is transmitted first. You should use this code to ensure that your checksums will match PCCOM’s.

- 6** Optionally, the first packet may contain the filename.
In the first (and only the first) packet, it is possible to send a data block containing the file’s name. If such a block is received by pccom or the PCCom library, then this filename will take precedence over the filename that was sent before the packets. Since the first filename may have been corrupted by noise, this provides a surer backup.

This data block should have the following data:

```

"!PCCom File Transfer Filename Block! " <no NULL>
<null-terminated file name>

```

The CRC for this block should be one higher than it would normally be—this signals that this block is of this special format.

- 7** Wait for a SYNC byte acknowledgment.
After each packet, you must make sure PCCOM responds positively with another SYNC byte. If the SYNC is not received, the packet likely failed. If the packet succeeded, continue sending packets as above (waiting for a SYNC after each) until done transferring the data.

If, instead of receiving a SYNC value, you receive a NAK_QUIT value, the target machine has aborted to an unrecoverable error, and there is little point in continuing to send data over the serial line.

When sending an optional filename packet, as described in step six, normally one does not re-send the packet if it fails to send.

- 8** Repeat steps five and seven as many times as necessary to transfer the entire file.
- 9** Send two zero bytes.
To make it absolutely clear that the file transfer has finished, send two zero bytes.

10.9

10.9.3.2 Retrieving a File Remotely

Retrieving a file from a machine running PCCOM is straightforward and uses the same file transfer protocol shown above for sending a file. The sequence of commands is different, however, and is listed below.

- 1** Notify PCCOM that you're getting a file.
Send the Get File escape character sequence to the serial port, notifying PCCOM that it should get ready to send a file. The escape sequence is <Esc>XF2.
- 2** Send the source file name.
Send the name of the file to be retrieved. The name is a string of sequential characters ending with a null byte. It may contain standard DOS wildcard characters.
- 3** Wait for SYNC byte signifying acknowledgment of the name transfer.
PCCOM will send a 0xFF character or a SYNC byte to acknowledge acceptance of the file name. If you do not receive this character, an error has likely occurred (e.g. the file does not exist).
- 4** Wait for another SYNC byte signifying file found.
PCCOM will send a SYNC byte to acknowledge that the desired file was found.
- 5** Receive destination file name.
This null-terminated string contains the place to write the file on the host machine.



6 Send a SYNC byte to acknowledge receipt of the destination file name. By sending a SYNC byte, the host acknowledges that it has received the destination file name and is ready to continue the transfer.

7 Receive the file size.
The next data received will be the file's size, expressed as a dword, with the low byte sent first.

8 Receive packets. 10.9
After you receive the file size from PCCOM, you will begin receiving data packets. Each packet you receive will follow the same format as described above for sending a file. Be aware of the BLOCK_QUOTE requirements when receiving the data (described in sending a file).

You will receive a checksum word at the end of each block's data. If the sent checksum matches the one you calculate from the received data, send a SYNC byte to the serial port (0xFF); otherwise send a NAK value. Calculate all your checksum values with the table and code presented in "Calculating Checksum Values," below.

When retrieving files, there will never be a file name packet such as described in step six of the file-sending procedure. Thus, you need not check to see whether a block whose CRC is off by one in fact contains a file name; you should not acknowledge this block.

9 Receive two zero bytes.
These make it clear that the file transfer has been completed.

10 Send an ACK byte.
This acknowledges the end of the file transfer.

10.9.3.3 Calculating Checksum Values

The CRC word that accompanies each block of transferred data must be calculated using the same code as PCCOM or you will probably have only unsuccessful transmissions. The code used by PCCOM is shown on the next page, and you may include it in your own file transfer program. PCSEND, PCGET, and PCS also use the same checksum calculation code. This checksum should be based only on the data itself; do not include the BLOCK_START, BLOCK_QUOTE, or BLOCK_END characters in your calculations.



Code Display 10-1 PCCOM Checksums

```

/*****
*                               CalcCRC                               *
*****/
* SUMMARY:      Calculate the CRC on a block of data.
* PASS:         char *buf      Pointer to the data buffer
*              short size      Size of the data buffer
*              short checksum  Previous checksum (0 at first)
10.9 * RETURN:      CRC value (2 bytes)
*****/

short  crcTable[] = {
    0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50a5, 0x60c6, 0x70e7,
    0x8108, 0x9129, 0xa14a, 0xb16b, 0xc18c, 0xd1ad, 0xe1ce, 0xf1ef,
    0x1231, 0x0210, 0x3273, 0x2252, 0x52b5, 0x4294, 0x72f7, 0x62d6,
    0x9339, 0x8318, 0xb37b, 0xa35a, 0xd3bd, 0xc39c, 0xf3ff, 0xe3de,
    0x2462, 0x3443, 0x0420, 0x1401, 0x64e6, 0x74c7, 0x44a4, 0x5485,
    0xa56a, 0xb54b, 0x8528, 0x9509, 0xe5ee, 0xf5cf, 0xc5ac, 0xd58d,
    0x3653, 0x2672, 0x1611, 0x0630, 0x76d7, 0x66f6, 0x5695, 0x46b4,
    0xb75b, 0xa77a, 0x9719, 0x8738, 0xf7df, 0xe7fe, 0xd79d, 0xc7bc,
    0x48c4, 0x58e5, 0x6886, 0x78a7, 0x0840, 0x1861, 0x2802, 0x3823,
    0xc9cc, 0xd9ed, 0xe98e, 0xf9af, 0x8948, 0x9969, 0xa90a, 0xb92b,
    0x5af5, 0x4ad4, 0x7ab7, 0x6a96, 0x1a71, 0x0a50, 0x3a33, 0x2a12,
    0xdbfd, 0xcbdc, 0xfbbf, 0xeb9e, 0x9b79, 0x8b58, 0xbb3b, 0xab1a,
    0x6ca6, 0x7c87, 0x4ce4, 0x5cc5, 0x2c22, 0x3c03, 0x0c60, 0x1c41,
    0xedae, 0xfd8f, 0xcdec, 0xddcd, 0xad2a, 0xbd0b, 0x8d68, 0x9d49,
    0x7e97, 0x6eb6, 0x5ed5, 0x4ef4, 0x3e13, 0x2e32, 0x1e51, 0x0e70,
    0xff9f, 0xefbe, 0xdfdd, 0xcffc, 0xbflb, 0xaf3a, 0x9f59, 0x8f78,
    0x9188, 0x81a9, 0xb1ca, 0xaleb, 0xd10c, 0xc12d, 0xf14e, 0xe16f,
    0x1080, 0x00a1, 0x30c2, 0x20e3, 0x5004, 0x4025, 0x7046, 0x6067,
    0x83b9, 0x9398, 0xa3fb, 0xb3da, 0xc33d, 0xd31c, 0xe37f, 0xf35e,
    0x02b1, 0x1290, 0x22f3, 0x32d2, 0x4235, 0x5214, 0x6277, 0x7256,
    0xb5ea, 0xa5cb, 0x95a8, 0x8589, 0xf56e, 0xe54f, 0xd52c, 0xc50d,
    0x34e2, 0x24c3, 0x14a0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
    0xa7db, 0xb7fa, 0x8799, 0x97b8, 0xe75f, 0xf77e, 0xc71d, 0xd73c,
    0x26d3, 0x36f2, 0x0691, 0x16b0, 0x6657, 0x7676, 0x4615, 0x5634,
    0xd94c, 0xc96d, 0xf90e, 0xe92f, 0x99c8, 0x89e9, 0xb98a, 0xa9ab,
    0x5844, 0x4865, 0x7806, 0x6827, 0x18c0, 0x08e1, 0x3882, 0x28a3,
    0xcb7d, 0xdb5c, 0xeb3f, 0xfb1e, 0x8bf9, 0x9bd8, 0xabbb, 0xbb9a,
    0x4a75, 0x5a54, 0x6a37, 0x7a16, 0x0af1, 0x1ad0, 0x2ab3, 0x3a92,
    0xfd2e, 0xed0f, 0xdd6c, 0xcd4d, 0xbdaa, 0xad8b, 0x9de8, 0x8dc9,

```



```

    0x7c26, 0x6c07, 0x5c64, 0x4c45, 0x3ca2, 0x2c83, 0x1ce0, 0x0cc1,
    0xef1f, 0xff3e, 0xcf5d, 0xdf7c, 0xaf9b, 0xbfba, 0x8fd9, 0x9ff8,
    0x6e17, 0x7e36, 0x4e55, 0x5e74, 0x2e93, 0x3eb2, 0x0ed1, 0x1ef0
};

unsigned short  IncCRC(unsigned short crc, char c){
    return ((crc << 8) ^ crcTable[((crc >> 8) ^ c) & 0xff]);
}

short          CalcCRC(char *buf, short size, short checksum){
/* The CRC is for the data part of the packet only.
* The CRC value is passed low byte first. */
    for (;size > 0; size--){
        checksum = IncCRC(checksum, *buf++);
    }
    return checksum;
}

```

10.10

10.10 pcget

Once you have **pccom** running on the target machine, you can invoke **pcget** on the development machine to yank a file from the target machine to the development machine.

Normally you would just type

```
pcget goemon.gif
```

to retrieve a file from the target machine's current directory or

```
pcget ..\clipart\jigen.gif
```

to retrieve it from another directory. To use different speed settings than the **pccom** tool would normally use (the **pccom** tool's settings are normally determined by the PTY environment variable), you may pass /b, /c, and /I flags, as you did when invoking the **pccom** tool. The /b flag sets the baud rate (e.g. "/b:19400"); the /c: flag sets the com port (e.g. "/c:3"), and the /I flag sets the IRQ level (e.g. "/I:3"). Note that these flags may be indicated with "-" instead of "/".

10.11 **pcs**

10.11

Once you have **pccom** running on the target machine, invoke **pcs** on the development machine to send your geode's executable to the proper place on the target machine. The **pcs** tool figures out which files to send (and which directory to send them to) via a number of rules set up in the `ROOT_DIR\INCLUDE\PCS.PAT` file (see Table 10-1 on page 464 for these rules). Thus it knows to send applications (files that end in `.GEO`) to the target machine's `WORLD` directory. (Note that the `/` and `-` characters in the flags below are interchangeable—hyphens are used with the `n`, `S`, and `t` flags by tradition.)

/I:IRQ Communicate at a different interrupt level than **pccom** is presently using (this is a dangerous option).

-S file Send all files mentioned in the file *file*.

/b:baudRate
Communicate at a different speed than **pccom** is presently using (this is a dangerous option).

/c:portNum
Communicate via a different COM port than **pccom** is presently using (this is a dangerous option).

-h Get help.

-n Send non-Error Checking version instead of EC.

-t token Send all files associated with the named token (see below).

When using a token, **pcs** looks up the token in the `INCLUDE\SEND` file to determine which files should be sent. Generally all executables associated with an application, library, or mechanism are sent when the appropriate token is passed. Look in the `SEND` file to find out what the accepted tokens are and what they send. Suppose your send file consisted of the following lines:



PC	DRIVER/VIDEO/DUMB/HGC/HGC	GEO
HGCAT	DRIVER/VIDEO/DUMB/HGC/HGC	GEO
PCB	DRIVER/VIDEO/DUMB/HGC/HGC	GEO
PCB	DRIVER/MOUSE/LOGIBUS/LOGIBUS	GEO
PCS	DRIVER/VIDEO/DUMB/HGC/HGC	GEO
PCS	DRIVER/MOUSE/LOGISER/LOGISER	GEO

Typing

`pcs -t pc`

10.11

would send the file DRIVER\VIDEO\DUMB\HGC\HGCEC.GEO (HGC.GEO if sending non-EC). Typing “`pcs -S pcb`” would send that file, and also DRIVER\MOUSE\LOGIBUS\LOGIBUSE.GEO (or LOGIBUS.GEO).

Using Tools

464

Table 10-1 *Destination of Files Sent by Pcs Development Directory*

	Suffix	Target Directory
*\LIBRARY\HWR\DATA\	*	PRIVDATA\HWR
\LIBRARY\SPELL\DICTS	*	USERDATA\DICTS
*\LIBRARY\SAVER\	GEO	WORLD
\LIBRARY\SAVER\	GEO	SYSTEM\SAVERS
\LIBRARY\PREF	GEO	SYSTEM\PREF
10.11 *\LIBRARY\TRANSLAT\GRAPHICS**	GEO	SYSTEM
*\LIBRARY\TRANSLAT\TEXT\MSMFILE\	GEO	SYSTEM
\LIBRARY\TRANSLAT	GEO	SYSTEM\IMPEX
\LIBRARY\FMTOOLS	GEO	SYSTEM\FILEMGR
\LIBRARY\NET	GEO	SYSTEM\SYSAPPL
\LIBRARY	GEO	SYSTEM
\DRIVER\IFS	GEO	SYSTEM\FS
\DRIVER\COMM	GEO	SYSTEM
\DRIVER\NET	GEO	SYSTEM\NET
\DRIVER\FAX	GEO	SYSTEM\FAX
\DRIVER\TASK	GEO	SYSTEM\TASK
\DRIVER\FONT	GEO	SYSTEM\FONT
\DRIVER\KEYBOARD	GEO	SYSTEM\KBD
\DRIVER\MOUSE	GEO	SYSTEM\MOUSE
\DRIVER\PRINTER	GEO	SYSTEM\PRINTER
\DRIVER\SWAP	GEO	SYSTEM\SWAP
\DRIVER\VIDEO	GEO	SYSTEM\VIDEO
\DRIVER\STREAM	GEO	SYSTEM
\DRIVER\POWER	GEO	SYSTEM\POWER
\DRIVER\SOUND	GEO	SYSTEM\SOUND
\APPL\LAUNCHER	GEO	PRIVDATA
\APPL\WELCOME	GEO	SYSTEM\SYSAPPL
\APPL\PREFEREN\SETUP	GEO	SYSTEM\SYSAPPL
\APPL\SDK_C	GEO	WORLD\C
\APPL\SDK_ASM	GEO	WORLD\ASM
\APPL	GEO	WORLD
\FONTDATA	FNT	USERDATA\FONT
\DOCUMENT	*	DOCUMENT
\DOSAPPL	*	.



10.12 pcsend

The **pcsend** tool sends files from the development machine to the target machine (assuming that the target machine is running pccom). Normally you use pcs to send geodes between machines, as that tool has knowledge of where geodes belong; pcsend is for those cases where the pcs tool's automatic behavior is undesirable. With pcsend, you can specify source and destination explicitly.

10.12

Normally you will invoke pcsend by typing something like:

```
pcsend zenigata.pcx
```

or (to send to a directory other than the target PC's current directory):

```
pcsend zenigata.pcx /d:..\clipart
```

In addition to the /d option, you may also pass /b, /c, or /I options to override the speed, port, and/or IRQ values in the PTTY environment variable. (You may use "-" instead of "/" when passing these flags.)

```
pcsend sendslow.com /b:2400
```

10.13 pmake

The **pmake** program is a make utility. This means that it takes a directory of sources and a makefile which contains knowledge of how to turn these sources into geodes. If there were no **pmake**, then you would have to type goc, bcc, and glue each time.

The **pmake** program will work correctly if you have set up your files correctly. This means that you must have a makefile. You should run **mkmf** to generate a makefile that knows how to generate geodes. If you are an experienced C programmer, you may have come up with some customizations that you use with your make utility. We still suggest that you work from a standard makefile (which knows about Goc and Glue), but include your customizations in a **local.mk** file (see section 10.13.2 of chapter 10).

When you have just added or removed source files from a geode, you will have to generate new dependency information, which **pmake** will use when doing

other makes. You can use **pmake** to generate this information (it will make an appropriate call to the **makedpnd** program; it will store the result in **depends.mk**):

```
pmake depend
```

Once you have created your source and make files, all you need to do to invoke **pmake** is type

```
pmake
```

10.13

The **pmake** tool can be used to construct a specific target. Thus, if you need to generate an .OBJ file but do not need the rest of the geode, you may type something like:

```
pmake sdtrk.obj
```

The system makefiles have been set up with knowledge of some special targets. If you are making a library and wish to install it in the proper directory so that all applications may use it, then signal this to **pmake**:

```
pmake lib
```

The **pmake** program does have command-line arguments, but these are used only rarely; they are detailed below.

10.13.1 Copyright Notice and Acknowledgment

The **pmake** tool comes under the following copyright notice:

Copyright© 1988, 1989 by the Regents of the University of California
Copyright© 1988, 1989 by Adam de Boor
Copyright© 1989 by Berkeley Softworks

Permission to use, copy, modify, and distribute this software (**pmake**) and its documentation for any non-commercial purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies. The University of California, Berkeley Softworks, and Adam de Boor make no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

The **pmake** program for the PC uses the SPAWNO routines by Ralf Brown to minimize memory use while shelling to DOS and running other programs.



10.13.2 How to Customize pmake

For most applications, executing **mkmf** will generate a perfect makefile. However, you may be creating an unusual geode or have some makefile definitions which you want to include. Fortunately, there are ways to customize your make environment without having to build a MAKEFILE from scratch.

Makefiles can `#include` other makefiles. If you have a file named LOCAL.MK in your make directory, then the standard makefile generated by mkmf will include it; if you wish to customize your makes, you can create a LOCAL.MK file and fill it with appropriate make directives. 10.13

Depending on how much customization you need to do, you may wish to read on to find out about makefile syntax. However, there are several simple things you can do without learning too much about makefiles.

```
# Pass extra flags to Goc:
GOCFLAGS      += flag1 flag2

# Pass extra flags to your C compiler:
CCOMFLAGS     += flag1 flag2

# Pass extra flags to the Esp assembler (if you have
# that tool):
ASMFLAGS      += flag1 flag2

# Pass extra flags to the Glue linker:
LINKFLAGS     += flag1 flag2

# Look somewhere special for .GOC files
# (This pattern applies to any suffix):
.PATH.GOC     : $(ROOT_DIR)\DIR1 \DIR2

# Specify geode name:
GEODE         = NAME

# Set NO_EC variable (which signals that we don't
# want to make an Error Checking version):
NO_EC         = 1

# If your preprocessor is not reachable via the
# Path environment variable:
CPP           = vol:\path\name
```



```
# If your C compiler is not reachable via the Path
# environment variable:
CCOM          = vol:\path\name
# Include some other make file
#include "OTHERMF.MK"
# Include the standard makefile directives
# This will include INCLUDE\GEODE.MK:
#include <$(SYSMAKEFILE)>
```

10.13

Code Display 10-2 Sample local.mk Files

```
---FINGER\LOCAL.MK
# Local Makefile for FPaint
# FPaint is stored in a directory called FINGER. This would normally confuse
# pmake, which expects the geode name to be the same as the directory name.
# Let us, therefore, alert pmake to the geode's real name:
GEODE          = FPAINT
# This was the only thing we wanted to change, so include standard definitions:
#include <$(SYSMAKEFILE)>

---PROMO\LOCAL.MK
# Local Makefile for Promo
# Promo uses some clip art that isn't in or below its source directory, so we
# tell pmake where to look for it:
.PATH.GOH : $(CSOURCE_PATHS) $(CINCLUDE_DIR) $(ROOT_DIR)\LOGOART
# This program contains no Error Checking code (See
# "GEOS Programming," Chapter 5 of the Concepts Book for information
# about EC code. So we tell pmake not to bother making an Error Checking
# version:
NO_EC = 1
# Include the standard system makefile:
#include <$(SYSMAKEFILE)>
```



10.13.3 Command Line Arguments

The **pmake** program comes with a wide variety of flags to choose from. They must be passed in the following order: flags (if any), variable assignments (if any), target (if any).

```
pmake [flags] [variables] [target]
```

The flags are as follows:

10.13

-d *info* This causes **pmake** to print out debugging information that may prove useful to you. The *info* parameter is a string of single characters that tell **pmake** what aspects you are interested in. Most of these options will make little sense to you unless you've dealt with Make before. Just remember where this table is and come back to it as you read on. The characters and the information they produce are as follows:

*	All debugging information.
c	Conditional evaluation.
d	The searching and caching of directories.
m	The making of each target: what target is being examined; when it was last modified; whether it is out-of-date; etc.
p	Makefile parsing.
r	Remote execution.
s	The application of suffix-transformation rules.
t	The maintenance of the list of targets.
v	Variable assignment.

Of these, the “m” and “s” flags will be most useful.

-f *file* Specify a makefile to read different from the default (MAKEFILE). If *file* is “-”, **pmake** uses the standard input. This is useful for making “quick and dirty” makefiles.

-h Prints out a summary of the various flags **pmake** accepts. It can also be used to find out what level of concurrence was compiled into the version of **pmake** you are using (look at -J and -L) and various other information on how **pmake** is configured.

- i** If you give this flag, **pmake** will ignore non-zero status returned by any of its shells. It's like placing a "-" before all the commands in the makefile.
- k** This is similar to -i in that it allows **pmake** to continue when it sees an error, but unlike -i where **pmake** continues blithely as if nothing went wrong, -k causes it to recognize the error and only continue work on those things that don't depend on the target, either directly or indirectly (through depending on something that depends on it), whose creation returned the error. (The "k" is for "keep going".)
- n** This flag tells **pmake** not to execute the commands needed to update the out-of-date targets in the makefile. Rather, **pmake** will simply print the commands it would have executed and exit. This is particularly useful for checking the correctness of a makefile. If **pmake** doesn't do what you expect it to, it's a good chance the makefile is wrong.
- p *number*** You should never have to use this option; it is used for debugging **pmake**. This causes **pmake** to print its input in a reasonable form, though not necessarily one that would make immediate sense to anyone but a **pmake** expert. The *number* is a bitwise-or of 1 and 2 where 1 means it should print the input before doing any processing and 2 says it should print it after everything has been re-created. Thus "-p 3" would print it twice—once before processing and once after (you might find the difference between the two interesting).
- q** If you give **pmake** this flag, it will not try to re-create anything. It will just query to see if anything is out-of-date and exit non-zero if so.
- s** This silences **pmake**, preventing it from printing commands before they're executed. It is the equivalent of putting an "@" before every command in the makefile.
- t** Rather than try to re-create a target, **pmake** will simply "touch" it so as to make it appear up-to-date. If the target didn't exist before, it will when **pmake** finishes, but if the target did exist, it will appear to have been updated.
- D *var*** Allows you to define a variable to have 1 as its value. The variable is a global variable, not a command-line variable. This is useful mostly for people who are used to the C compiler arguments and those using conditionals, which are described in section 10.13.5.2 on page 495, below.
- I *directory*** Tells **pmake** another place to search for included makefiles.



- W** Suppresses **pmake**'s warnings. Note that tools which **pmake** invokes (Gcc, Glue, etc.) may still print out warnings of their own.

10.13.4 Contents of a Makefile

The **pmake** program takes as input a file that tells

- ◆ which files depend on which other files and
- ◆ what to do about files that are “out-of-date.”

10.13

This file is known as a “makefile” and is usually kept in the top-most directory of the system to be built. While you can call the makefile anything you want, **pmake** will look for MAKEFILE in the current directory if you don't tell it otherwise. To specify a different makefile, use the **-f** flag (e.g. “**pmake -f program.mk**”).

A makefile has four different types of lines in it:

- ◆ File dependency specifications
- ◆ Creation commands
- ◆ Variable assignments
- ◆ Comments, include statements and conditional directives

Any line may be continued over multiple lines by ending it with a backslash (“\”). The backslash, following newline and any initial whitespace on the following line are compressed into a single space before the input line is examined by **pmake**.

10.13.4.1 Dependency Lines

In any system, there are dependencies between the files that make up the system. For instance, in a program made up of several C source files and one header file, the C files will need to be re-compiled should the header file be changed. For a document of several chapters and one macro file, the chapters will need to be reprocessed if any of the macros changes. These are dependencies and are specified by means of dependency lines in the makefile.

On a dependency line, there are *targets* and *sources*, separated by a one- or two-character operator. The targets “depend” on the sources and are usually created from them. Any number of targets and sources may be specified on a dependency line. All the targets in the line are made to depend on all the sources. Targets and sources need not be actual files, but every source must be either an actual file or another target in the makefile. If you run out of room, use a backslash at the end of the line to continue onto the next one.

10.13

Any file may be a target and any file may be a source, but the relationship between the two (or however many) is determined by the “operator” that separates them. Three types of operators exist: one specifies that the datedness of a target is determined by the state of its sources, while another specifies other files (the sources) that need to be dealt with before the target can be re-created. The third operator is very similar to the first, with the additional condition that the target is out-of-date if it has no sources. These operations are represented by the colon, the exclamation point and the double-colon, respectively, and are mutually exclusive (to represent a colon in a target, you must precede it with a backslash: “\:”). Their exact semantics are as follows:

- : If a colon is used, a target on the line is considered to be “out-of-date” (and in need of creation) if
 - ◆ the target doesn’t exist or
 - ◆ any of the sources has been modified more recently than the target.

Under this operator, steps will be taken to re-create the target only if it is found to be out-of-date by using these two rules.

- ! If an exclamation point is used, the target will always be re-created, but this will not happen until all of its sources have been examined and re-created, if necessary.

- :: If a double-colon is used, a target is out-of-date if
 - ◆ the target has no sources,
 - ◆ any of the sources has been modified more recently than the target, or
 - ◆ the target doesn’t exist.

If the target is out-of-date according to these rules, it will be re-created. This operator also does something else to the targets, as described in section 10.13.4.2 on page 474).



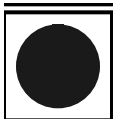
Suppose there are three C files (**a.c**, **b.c** and **c.c**) each of which includes the file **defs.h**. The dependencies between the files could then be expressed as follows:

```
PROGRAM.EXE      : A.OBJ B.OBJ C.OBJ
A.OBJ B.OBJ C.OBJ : DEFS.H
A.OBJ            : A.C
B.OBJ            : B.C
C.OBJ            : C.C
```

10.13

You may be wondering at this point, where A.OBJ, B.OBJ and C.OBJ came in and why they depend on defs.h and the C files don't. The reason is quite simple: PROGRAM.EXE cannot be made by linking together .c files—it must be made from .obj files. Likewise, if you change DEFS.H, it isn't the .c files that need to be re-created, it's the .obj files. If you think of dependencies in these terms—which files (targets) need to be created from which files (sources)—you should have no problems.

An important thing to notice about the above example is that all the .obj files appear as targets on more than one line. This is perfectly all right: the target is made to depend on all the sources mentioned on all the dependency lines. For example, A.OBJ depends on both DEFS.H and A.C.



Remember

Order of dependency lines is significant.

The order of the dependency lines in the makefile is important: the first target on the first dependency line in the makefile will be the one that gets made if you don't say otherwise. That's why PROGRAM.EXE comes first in the example makefile, above.

Both targets and sources may contain the standard C-Shell wildcard characters ({, }, *, ?, [, and]), but the square braces may only appear in the final component (the file portion) of the target or source. The characters mean the following things:

{ }

These enclose a comma-separated list of options and cause the pattern to be expanded once for each element of the list. Each expansion contains a different element. For example,

```
SRC\{WHIFFLE,BEEP,FISH}.C
```

expands to the three words "SRC\WHIFFLE.C", "SRC\BEEP.C", and "SRC\FISH.C". These braces may be nested and, unlike the other wildcard characters, the resulting words need not be actual files. All

other wildcard characters are expanded using the files that exist when **pmake** is started.

* This matches zero or more characters of any sort.

`SRC*.C`

will expand to the same three words as above as long as **src** contains those three files (and no other files that end in .c).

10.13

? Matches any single character.

[] This is known as a character class and contains either a list of single characters, or a series of character ranges ([a-z], for example means all characters between a and z), or both. It matches any single character contained in the list. E.g. [A-Za-z] will match all letters, while [0123456789] will match all numbers.

10.13.4.2 Shell Commands

At this point, you may be wondering how files are re-created. The re-creation is accomplished by commands you place in the makefile. These commands are passed to the shell to be executed and are expected to do what's necessary to update the target file. (The **pmake** program doesn't actually check to see if the target was created. It just assumes it's there.)

Shell commands in a makefile look a lot like shell commands you would type, with one important exception: each command in a makefile *must* be preceded by at least one tab.

Each target has associated with it a shell script made up of one or more of these shell commands. The creation script for a target should immediately follow the dependency line for that target. While any given target may appear on more than one dependency line, only one of these dependency lines may be followed by a creation script, unless the "::" operator was used on the dependency line.

If the double-colon was used, each dependency line for the target may be followed by a shell script. That script will only be executed if the target on the associated dependency line is out-of-date with respect to the sources on that line, according to the rules given earlier.



To expand on the earlier makefile, you might add commands as follows:

```
PROGRAM.EXE : A.OBJ B.OBJ C.OBJ
    BCC A.OBJ B.OBJ C.OBJ -o PROGRAM.EXE
A.OBJ B.OBJ C.OBJ : DEFS.H
A.OBJ : A.C
    bcc -c A.C
B.OBJ : B.C
    bcc -c B.C
C.OBJ : C.C
    bcc -c C.C
```

10.13

Something you should remember when writing a makefile is that the commands will be executed if the *target* on the dependency line is out-of-date, not the sources. In this example, the command “bcc -c a.c” will be executed if **a.obj** is out-of-date. Because of the “:” operator, this means that should **a.c** or **defs.h** have been modified more recently than **a.obj**, the command will be executed (**a.obj** will be considered out-of-date).

There is another way in which makefile commands differ from regular shell commands. The first two characters after the initial whitespace are treated specially. If they are any combination of “@” and “-”, they cause **pmake** to do things differently.

In most cases, shell commands are printed before they’re actually executed. This is to keep you informed of what’s going on. If an “@” appears, however, this echoing is suppressed. In the case of an **echo** command, perhaps “echo Linking index” it would be rather messy to output

```
echo Linking index
Linking index
```

The other special character is the dash (“-”). Shell commands finish with a certain “exit status.” This status is made available by the operating system to whatever program invoked the command. Normally this status will be zero if everything went ok and non-zero if something went wrong. For this reason, **pmake** will consider an error to have occurred if one of the shells it invokes returns a non-zero status. When it detects an error, **pmake**’s usual action is to abort whatever it’s doing and exit with a non-zero status itself. This behavior can be altered, however, by placing a “-” at the front of a command

(e.g. “-copy index index.old”) . In such a case, the non-zero status is simply ignored and **pmake** keeps going.

If the system call should be made through the DOS COMMAND.COM, precede the shell command with a backquote (`).

10.13.4.3 Variables

10.13

The **pmake** program has the ability to save text in variables to be recalled later at your convenience. Variables in **pmake** are used much like variables in the shell and, by tradition, consist of all upper-case letters. Variables are assigned using lines of the form

```
VARIABLE = value
```

append using lines of the form

```
VARIABLE += value
```

conditionally assigned (if the variable isn't already defined) by using lines of the form

```
VARIABLE ?= value
```

and assigned with expansion (i.e. the value is expanded (see below) before being assigned to the variable—useful for placing a value at the beginning of a variable, or other things) by using lines of the form

```
VARIABLE := value
```

Any whitespace before *value* is stripped off. When appending, a space is placed between the old value and the values being appended.

The final way a variable may be assigned is using lines of the form

```
VARIABLE != shell-command
```

or, if the shell command requires the use of the command.com interpreter,

```
VARIABLE != `shell-command`
```

In this case, *shell-command* has all its variables expanded (see below) and is passed off to a shell to execute. The output of the shell is then placed in the variable. Any newlines (other than the final one) are replaced by spaces



before the assignment is made. This is typically used to find the current directory via a line like:

```
CURRENT_DIR != `cd
```

The value of a variable may be retrieved by enclosing the variable name in parentheses or curly braces and preceding the whole thing with a dollar sign. For example, to set the variable CFLAGS to the string “-I\NIHON\LIB\LIBC -O”, you would place a line

```
CFLAGS = -I\NIHON\LIB\LIBC -O
```

10.13

in the makefile and use the expression

```
$(CFLAGS)
```

wherever you would like the string “-I\NIHON\LIB\LIBC -O” to appear. This is called variable expansion.

There are two different times at which variable expansion occurs: When parsing a dependency line, the expansion occurs immediately upon reading the line. Variables in shell commands are expanded when the command is executed. Variables used inside another variable are expanded whenever the outer variable is expanded (the expansion of an inner variable has no effect on the outer variable. That is, if the outer variable is used on a dependency line and in a shell command, and the inner variable changes value between when the dependency line is read and the shell command is executed, two different values will be substituted for the outer variable).

Variables come in four flavors, though they are all expanded the same and all look about the same. They are (in order of expanding scope)

- ◆ local variables.
- ◆ command-line variables.
- ◆ global variables.
- ◆ environment variables.

The classification of variables doesn't matter much, except that the classes are searched from the top (local) to the bottom (environment) when looking up a variable. The first one found wins.

Local Variables

Each target can have as many as seven local variables. These are variables that are only “visible” within that target’s shell script and contain such things as the target’s name, all of its sources (from all its dependency lines), those sources that were out-of-date, etc. Four local variables are defined for all targets. They are

10.13

.TARGET	The name of the target.
.OODATE	The list of the sources for the target that were considered out-of-date. The order in the list is not guaranteed to be the same as the order in which the dependencies were given.
.ALLSRC	The list of all sources for this target in the order in which they were given.
.PREFIX	The target without its suffix and without any leading path. For example, for the target <code>..\..\LIB\FSREAD.C</code> , this variable would contain <code>FSREAD</code> .

One other local variable, `.IMPSRC`, is set only for certain targets under special circumstances. It is discussed below.

Two of these variables may be used in sources as well as in shell scripts. These are `.TARGET` and `.PREFIX`. The variables in the sources are expanded once for each target on the dependency line, providing what is known as a “dynamic source,” allowing you to specify several dependency lines at once. For example,

```
$(OBSJ) : $(.PREFIX).c
```

will create a dependency between each object file and its corresponding C source file.

Command-line Variables

Command-line variables are set when **pmake** is first invoked by giving a variable assignment as one of the arguments. For example,

```
pmake "CFLAGS = -I\NIHON\LIB\LIBC -O"
```

would make `CFLAGS` be a command-line variable with the given value. Any assignments to `CFLAGS` in the makefile will have no effect, because once it is set, there is (almost) nothing you can do to change a command-line variable. Command-line variables may be set using any of the four assignment operators, though only `=` and `?:=` behave as you would expect them to, mostly



because assignments to command-line variables are performed before the makefile is read, thus the values set in the makefile are unavailable at the time. += is the same as = because the old value of the variable is sought only in the scope in which the assignment is taking place. The := and ?= operators will work if the only variables used are in the environment.

Global Variables

Global variables are those set or appended in the makefile. There are two classes of global variables: those you set and those **pmake** sets. The ones you set can have any name you want them to have, except they may not contain a colon or an exclamation point. The variables **pmake** sets (almost) always begin with a period and contain only upper-case letters. The variables are as follows: 10.13

.PMAKE The name by which **pmake** was invoked is stored in this variable. For compatibility, the name is also stored in the MAKE variable.

.MAKEFLAGS All the relevant flags with which **pmake** was invoked. This does not include such things as “-f” or variable assignments. Again for compatibility, this value is stored in the MFLAGS variable as well.

Two other variables, **.INCLUDES** and **.LIBS**, are covered in the section on special targets (See section 10.13.4.9 of chapter 10).

Global variables may be deleted using lines of the form:

```
#undef variable
```

The “#” must be the first character on the line. Note that this may only be done to global variables.

Environment Variables

Environment variables are passed by the shell that invoked **pmake** and are given by **pmake** to each shell it invokes. They are expanded like any other variable, but they cannot be altered in any way.

One special environment variable, **PMAKE**, is examined by **pmake** for command-line flags, variable assignments, etc. that it should always use. This variable is examined before the actual arguments to **pmake** are. In addition, all flags given to **pmake**, either through the **PMAKE** variable or on

the command line, are placed in this environment variable and exported to each shell **pmake** executes. Thus recursive invocations of **pmake** automatically receive the same flags as the top-most one.

Many other standard environment variables are defined and described in the Include\GEOS.MK included Makefile.

Using all these variables, you can compress the sample makefile even more:

10.13

```
OBJS = A.OBJ B.OBJ C.OBJ
PROGRAM.EXE : $(OBJS)
            BCC $(.ALLSRC) -o $(.TARGET)
$(OBJS) : DEFS.H
A.OBJ : A.C
            BCC -c A.C
B.OBJ : B.C
            BCC -c B.C
C.OBJ : C.C
            BCC -c C.C
```

In addition to variables which **pmake** will use, you can set environment variables which shell commands may use using the `pmake_set` directive.

```
.C.EBJ :
pmake_set CL = $(CCOMFLAGS) /Fo$(.TARGET)
$(CCOM) $(.IMPSRC)
```

You might use the above sequence to set up an argument list in the CL environment variable if your compiler (invoked with CCOM) needed its arguments in such a variable and was unable to take arguments in a file.

10.13.4.4 Comments

Comments in a makefile start with a “#” character and extend to the end of the line. They may appear anywhere you want them, except where they might be misinterpreted as a shell command.



10.13.4.5 Transformation Rules

As you know, a file's name consists of two parts: a base name, which gives some hint as to the contents of the file, and a suffix, which usually indicates the format of the file. Over the years, as DOS has developed, naming conventions, with regard to suffixes, have also developed that have become almost incontrovertible. For example, a file ending in .C is assumed to contain C source code; one with a .OBJ suffix is assumed to be a compiled, relocatable object file that may be linked into any program. One of the best aspects of **pmake** comes from its understanding of how the suffix of a file pertains to its contents and their ability to do things with a file based solely on its suffix. This ability comes from something known as a transformation rule. A transformation rule specifies how to change a file with one suffix into a file with another suffix.

10.13

A transformation rule looks much like a dependency line, except the target is made of two known suffixes stuck together. Suffixes are made known to **pmake** by placing them as sources on a dependency line whose target is the special target .SUFFIXES. For example:

```
.SUFFIXES          : .obj .c
.c.obj             :
                   $(CCOM) $(CFLAGS) -c $(.IMPSRC)
```

The creation script attached to the target is used to transform a file with the first suffix (in this case, .c) into a file with the second suffix (here, .obj). In addition, the target inherits whatever attributes have been applied to the transformation rule. The simple rule above says that to transform a C source file into an object file, you compile it using your C compiler with the -c flag.

This rule is taken straight from the system makefile. Many transformation rules (and suffixes) are defined there; you should look there for more examples (type “pmake -h” to find out where it is).

There are some things to note about the transformation rule given above:

- 1 The .IMPSRC variable. This variable is set to the “implied source” (the file from which the target is being created; the one with the first suffix), which, in this case, is the .c file.
- 2 The CFLAGS variable. Almost all of the transformation rules in the system makefile are set up using variables that you can alter in your

makefile to tailor the rule to your needs. In this case, if you want all your C files to be compiled with the `-g` flag, to provide information for Swat or CodeView, you would set the `CFLAGS` variable to contain `-g` ("`CFLAGS = -g`") and **pmake** would take care of the rest.

To give you a quick example, the makefile could be changed to this:

10.13

```
OBJS = A.OBJ B.OBJ C.OBJ
PROGRAM .EXE      : $(OBJS)
$(CCOM) -o $(.TARGET) $(.ALLSRC)
$(OBJS)           : DEFS.H
```

The transformation rule given above takes the place of the 6 lines.

```
A.OBJ : A.C
      BCC -c A.C
B.OBJ : B.C
      BCC -c B.C
C.OBJ : C.C
      BCC -c C.C
```

Now you may be wondering about the dependency between the `.obj` and `.c` files—it's not mentioned anywhere in the new makefile. This is because it isn't needed: one of the effects of applying a transformation rule is the target comes to depend on the implied source (hence the name).

For a more detailed example, Suppose you have a makefile like this:

```
A.EXE      : A.OBJ B.OBJ
$(CCOM) $(.ALLSRC)
```

and a directory set up like this:

total 4

MAKEFILE	34	09-07-89	12:43a
A	C	119	10-03-89
A	OBJ	201	09-07-89
B	C	69	09-07-89



While just typing “**pmake**” will do the right thing, it’s much more informative to type “**pmake -ds**” This will show you what **pmake** is up to as it processes the files. In this case, **pmake** prints the following:

```
Suff_FindDeps (A.EXE)
    using existing source A.OBJ
    applying .OBJ -> .EXE to "A.OBJ"
Suff_FindDeps (A.OBJ)
    trying A.C...got it
    applying .C -> .OBJ to "A.C"
Suff_FindDeps (B.OBJ)
    trying B.C...got it
    applying .C -> .OBJ to "B.C"
Suff_FindDeps (A.C)
    trying A.Y...not there
    trying A.L...not there
    trying A.C,V...not there
    trying A.Y,V...not there
    trying A.L,V...not there
Suff_FindDeps (B.C)
    trying B.Y...not there
    trying B.L...not there
    trying B.C,V...not there
    trying B.Y,V...not there
    trying B.L,V...not there
--- A.OBJ ---
bcc -c A.C
--- B.OBJ ---
bcc -c B.C
--- A.EXE ---
bcc A.OBJ B.OBJ
```

10.13

Suff_FindDeps is the name of a function in **pmake** that is called to check for implied sources for a target using transformation rules. The transformations it tries are, naturally enough, limited to the ones that have been defined (a transformation may be defined multiple times, by the way, but only the most recent one will be used). You will notice, however, that there is a definite order to the suffixes that are tried. This order is set by the relative positions

of the suffixes on the `.SUFFIXES` line—the earlier a suffix appears, the earlier it is checked as the source of a transformation. Once a suffix has been defined, the only way to change its position is to remove all the suffixes (by having a `.SUFFIXES` dependency line with no sources) and redefine them in the order you want. (Previously-defined transformation rules will be automatically redefined as the suffixes they involve are re-entered.)

10.13

Another way to affect the search order is to make the dependency explicit. In the above example, `a.exe` depends on `a.obj` and `b.obj`. Since a transformation exists from `.obj` to `.exe`, **pmake** uses that, as indicated by the “using existing source `a.obj`” message.

The search for a transformation starts from the suffix of the target and continues through all the defined transformations, in the order dictated by the suffix ranking, until an existing file with the same base (the target name minus the suffix and any leading directories) is found. At that point, one or more transformation rules will have been found to change the one existing file into the target.

For example, ignoring what’s in the system makefile for now, say you have a makefile like this:

```
.SUFFIXES : .EXE .OBJ .C .Y .L
.L.C :
    LEX $(.IMPSRC)
    MOVE LEX.YY.C $(.TARGET)
.Y.C :
    YACC $(.IMPSRC)
    MOVE Y.TAB.C $(.TARGET)
.C.OBJ :
    BCC -L $(.IMPSRC)
.OBJ.EXE :
    BCC -o $(.TARGET) $(.IMPSRC)
```

and the single file `jive.l`. If you were to type **pmake -rd ms jive.exe**, you would get the following output for `jive.exe`:

```
Suff_FindDeps (JIVE.EXE)
trying JIVE.OBJ...not there
trying JIVE.C...not there
trying JIVE.Y...not there
```



```

trying JIVE.L...got it
applying .L -> .C to "JIVE.L"
applying .C -> .OBJ to "JIVE.C"
applying .OBJ -> .EXE to "JIVE.OBJ"

```

The **pmake** tool starts with the target `jive.exe`, figures out its suffix (`.exe`) and looks for things it can transform to a `.exe` file. In this case, it only finds `.obj`, so it looks for the file `JIVE.OBJ`.

It fails to find it, so it looks for transformations into a `.obj` file. Again it has only one choice: `.c`. So it looks for `JIVE.C` and fails to find it. At this point it can create the `.c` file from either a `.y` file or a `.l` file. Since `.y` came first on the `.SUFFIXES` line, it checks for `jive.y` first, but can't find it, so it looks for `jive.l`. At this point, it has defined a transformation path as follows: `.l->.c->.obj->.exe` and applies the transformation rules accordingly. For completeness, and to give you a better idea of what **pmake** actually did with this three-step transformation, this is what **pmake** printed for the rest of the process:

10.13

```

Suff_FindDeps (JIVE.OBJ)
using existing source JIVE.C
applying .C -> .OBJ to "JIVE.C"
Suff_FindDeps (JIVE.C)
using existing source JIVE.L
applying .L -> .C to "JIVE.L"
Suff_FindDeps (JIVE.L)
Examining JIVE.L...modified 17:16:01 Oct 4,
  1987...up-to-date
Examining JIVE.C...non-existent...out-of-date
--- JIVE.C ---
LEX JIVE.L
...meaningless lex output deleted...
MV LEX.YY.C JIVE.C
Examining JIVE.OBJ...non-existent...out-of-date
--- JIVE.OBJ ---
bcc -c JIVE.C
Examining JIVE.EXE...non-existent...out-of-date
--- JIVE.EXE ---
bcc -o JIVE.EXE JIVE.OBJ

```

10.13.4.6 Including Other Makefiles

Just as for programs, it is often useful to extract certain parts of a makefile into another file and just include it in other makefiles somehow. Many compilers allow you to use something like

```
#include "defs.h"
```

10.13

to include the contents of defs.h in the source file. The **pmake** program allows you to do the same thing for makefiles, with the added ability to use variables in the filenames. An include directive in a makefile looks either like this

```
#include <file>
```

or like this

```
#include "file"
```

The difference between the two is where **pmake** searches for the file: the first way, **pmake** will look for the file *only* in the system makefile directory (to find out what that directory is, give **pmake** the -h flag).

For files in double-quotes, the search is more complex; **pmake** will look in the following places in the given order:

- 1 The directory of the makefile that's including the file.
- 2 The current directory (the one in which you invoked **pmake**).
- 3 The directories given by you using -I flags, in the order in which you gave them.
- 4 Directories given by .PATH dependency lines.
- 5 The system makefile directory.

You are free to use **pmake** variables in the filename—**pmake** will expand them before searching for the file. You must specify the searching method with either angle brackets or double-quotes *outside* of a variable expansion. That is, the following

```
SYSTEM= <command.mk>  
#include $(SYSTEM)
```



won't work; instead use the following:

```
SYSTEM= command.mk
#include <$(SYSTEM)>
```

10.13.4.7 Saving Commands

There may come a time when you will want to save certain commands to be executed when everything else is done, by inserting an ellipsis “...” in the Makefile. Commands saved in this manner are only executed if **pmake** manages to re-create everything without an error.

10.13

10.13.4.8 Target Attributes

The **pmake** tool allows you to give attributes to targets by means of special sources. Like everything else **pmake** uses, these sources begin with a period and are made up of all upper-case letters. By placing one (or more) of these as a source on a dependency line, you are “marking” the target(s) with that attribute.

Any attributes given as sources for a transformation rule are applied to the target of the transformation rule when the rule is applied.

- .DONTCARE** If a target is marked with this attribute and **pmake** can't figure out how to create it, it will ignore this fact and assume the file isn't really needed or actually exists and **pmake** just can't find it. This may prove wrong, but the error will be noted later on, not when **pmake** tries to create the target so marked. This attribute also prevents **pmake** from attempting to touch the target if given the “-t” flag.
- .EXEC** This attribute causes its shell script to be executed while having no effect on targets that depend on it. This makes the target into a sort of subroutine. EXEC sources don't appear in the local variables of targets that depend on them (nor are they touched if **pmake** is given the -t flag).
- .IGNORE** Giving a target the .IGNORE attribute causes **pmake** to ignore errors from any of the target's commands, as if they all had “-” before them.
- .MAKE** The .MAKE attribute marks its target as being a recursive invocation of **pmake**. This forces **pmake** to execute the script associated with the target (if it's out-of-date) even if you gave the -n or -t flag. By doing this, you can start at the top of a system and type

```
pmake -n
```

and have it descend the directory tree (if your makefiles are set up correctly), printing what it would have executed if you hadn't included the `-n` flag.

10.13

.NOTMAIN Normally, if you do not specify a target to make in any other way, **pmake** will take the first target on the first dependency line of a makefile as the target to create. That target is known as the “Main Target” and is labeled as such if you print the dependencies out using the `-p` flag. Giving a target, this attribute tells **pmake** that the target is definitely not the Main Target. This allows you to place targets in an included makefile and have **pmake** create something else by default.

.PRECIOUS When **pmake** is interrupted (by someone typing control-C at the keyboard), it will attempt to clean up after itself by removing any half-made targets. If a target has the **.PRECIOUS** attribute, however, **pmake** will leave it alone. A side effect of the “`::`” operator is to mark the targets as **.PRECIOUS**.

.SILENT Marking a target with this attribute keeps its commands from being printed when they're executed, just as if they had an “`@`” in front of them.

.USE By giving a target this attribute, you turn it into **pmake**'s equivalent of a macro. When the target is used as a source for another target, the other target acquires the commands, sources and attributes (except **.USE**) of the source. If the target already has commands, the **.USE** target's commands are added to the end. If more than one **.USE**-marked source is given to a target, the rules are applied sequentially.

The typical **.USE** rule will use the sources of the target to which it is applied (as stored in the **.ALLSRC** variable for the target) as its “arguments.” Several system makefiles (not to be confused with *the* system makefile) make use of these **.USE** rules to make developing easier (they're in the default, system makefile directory).

10.13.4.9 Special Targets

There are certain targets that have special meaning to **pmake**. When you use one on a dependency line, it is the only target that may appear on the left-hand-side of the operator. As for the attributes and variables, all the special targets begin with a period and consist of upper-case letters only. The targets are as follows:



- .BEGIN Any commands attached to this target are executed before anything else is done. You can use it for any initialization that needs doing.
- .DEFAULT This is sort of a .USE rule for any target (that was used only as a source) that **pmake** can't figure out any other way to create. It's only "sort of" a .USE rule because only the shell script attached to the .DEFAULT target is used. The .IMPSRC variable of a target that inherits .DEFAULT's commands is set to the target's own name.
- .END This serves a function similar to .BEGIN, in that commands attached to it are executed once everything has been re-created (so long as no errors occurred). It also serves the extra function of being a place on which **pmake** can hang commands you put off to the end. Thus the script for this target will be executed before any of the commands you save with the ellipsis marker. 10.13
- .IGNORE This target marks each of its sources with the .IGNORE attribute. If you don't give it any sources, then it is like giving the -i flag when you invoke **pmake**—errors are ignored for all commands.
- .INCLUDES The sources for this target are taken to be suffixes that indicate a file that can be included in a program source file. The suffix must already be declared with .SUFFIXES. Any suffix so marked will have the directories on its search path (see .PATH, below) placed in the .INCLUDES variable, each preceded by a "-I" flag. This variable can then be used as an argument for the compiler in the normal fashion. The ".h" suffix is already marked in this way in the system makefile. For example, if you have


```
.SUFFIXES          : .PCX
.PATH.PCX          : \CLIPART
.INCLUDES          : .PCX
```

pmake places "-I\CLIPART" in the .INCLUDES variable and you can say

```
bcc $(.INCLUDES) -c xprogram.c
```

(Note: the .INCLUDES variable is not actually filled in until the entire makefile has been read.)
- .INTERRUPT When **pmake** is interrupted, it will execute the commands in the script for this target, if it exists.
- .LIBS This does for libraries what .INCLUDES does for include files, except the flag used is "-L", as required by those linkers that allow you to tell them where to find libraries. The variable used is .LIBS.

- .MAIN** If you didn't give a target (or targets) to create when you invoked **pmake**, it will take the sources of this target as the targets to create.
- .MAKEFLAGS** This target provides a way for you to always specify flags for **pmake** when the makefile is used. The flags are just as they would be typed to the shell (except you can't use shell variables unless they're in the environment), though the **-f** and **-r** flags have no effect.
- .PATH** If you give sources for this target, **pmake** will take them as directories in which to search for files it cannot find in the current directory. If you give no sources, it will clear out any directories added to the search path before.
- .PATHsuffix** This does a similar thing to **.PATH**, but it does it only for files with the given suffix. The suffix must have been defined already. Look at section 10.13.5.1 on page 493 for more information.
- .PRECIOUS** Similar to **.IGNORE**, this gives the **.PRECIOUS** attribute to each source on the dependency line, unless there are no sources, in which case the **.PRECIOUS** attribute is given to every target in the file.
- .RECURSIVE** This target applies the **.MAKE** attribute to all its sources. It does nothing if you don't give it any sources.
- .SILENT** When you use **.SILENT** as a target, it applies the **.SILENT** attribute to each of its sources. If there are no sources on the dependency line, then it is as if you gave **pmake** the **-s** flag and no commands will be echoed.
- .SUFFIXES** This is used to give new file suffixes for **pmake** to handle. Each source is a suffix **pmake** should recognize. If you give a **.SUFFIXES** dependency line with no sources, **pmake** will forget about all the suffixes it knew.

In addition to these targets, a line of the form

```
attribute : sources
```

applies the attribute to all the targets listed as sources .

10.13.4.10 Modifying Variable Expansion

Variables need not always be expanded verbatim. The **pmake** program defines several modifiers that may be applied to a variable's value before it is expanded. You apply a modifier by placing it after the variable name with a colon between the two, like so:

```
${VARIABLE:modifier}
```



Each modifier is a single character followed by something specific to the modifier itself. You may apply as many modifiers as you want—each one is applied to the result of the previous and is separated from the previous by another colon.

There are several ways to modify a variable's expansion:

:Mpattern This is used to select only those words (a word is a series of characters that are neither spaces nor tabs) that match the given pattern. The pattern is a wildcard pattern like that used 10.13 by the shell, where `*` means zero or more characters of any sort; `?` is any single character; `[abcd]` matches any single character that is one of “a”, “b”, “c” or “d” (there may be any number of characters between the brackets); `[0-9]` matches any single character that is between “0” and “9” (i.e. any digit. This form may be freely mixed with the other bracket form), and `\` is used to escape any of the characters “*”, “?”, “[” or “.”, leaving them as regular characters to match themselves in a word.

Remember that the pattern matcher requires you to prefix certain characters with a backslash, including backslash itself. this can lead to some impressive search strings, because **pmake** also requires that backslashes be preceded with backslashes:

```
#if !empty(CURRENT_DIR:M*\\APPL\\*)
```

The above line checks to see if the current directory matches the form “*\\APPL*”. (The pattern matcher is passed the string “*\\APPL*”.)

:Npattern This is identical to `:M` except it substitutes all words that don't match the given pattern.

:Xpattern This is like `:M` except that it returns only part of the matching string. You mark which part of the string you are interested in by enclosing it within backslashed square brackets (“\[” and “\]”) (however, due to the backslash rules, you must actually use “\\[” and “\\]”).

```
DEVEL_DIR := \
$(CURRENT_DIR:X\[*\\$(ROOT_DIR:T)\\*\\]\\\\*)
```

The above line returns part of the `CURRENT_DIR` string, specifically the directory just under the root directory. Free of backslashes, it's searching for `[*\\$(ROOT_DIR:T)*]`. If there

is a subdirectory below the development directory, then this will strip off the lower layers.

:S/search-string/replacement-string/[g]

This causes the first occurrence of *search-string* in the variable to be replaced by *replacement-string*, unless the “g” flag is given at the end, in which case all occurrences of the string are replaced. The substitution is performed on each word in the variable in turn. If search-string begins with a caret (“^”), the string must match starting at the beginning of the word. If search-string ends with a dollar sign (“\$”), the string must match to the end of the word (these two may be combined to force an exact match). If a backslash precedes these two characters, however, they lose their special meaning. Variable expansion also occurs in the normal fashion inside both the search-string and the replacement-string, except that a backslash is used to prevent the expansion of a “\$”, not another dollar sign, as is usual. Note that search-string is just a string, not a pattern, so none of the usual regular-expression/wildcard characters have any special meaning save “^” and “\$”. In the replacement string, the “&” character is replaced by the search-string unless it is preceded by a backslash. Thus, “:S/[A-D]/&&/'” will automatically cause any of the characters between “A” and “D” to be repeated. You are allowed to use any character except colon or exclamation point to separate the two strings. This so-called delimiter character may be placed in either string by preceding it with a backslash.

10.13

:T

Replaces each word in the variable expansion by its last component (its “tail”). For example, given

```
OBJS = ..\LIB\A.OBJ B \USR\LIB\LIBM.A
TAILS = $(OBJS:T)
```

the variable TAILS would expand to “a.obj b libm.a”.

:H

This is similar to :T, except that every word is replaced by everything but the tail (the “head”). Using the same definition of OBJS, the string “\$(OBJS:H)” would expand to “..\LIB\USR\LIB” Note that the final slash on the heads is removed and anything without a head is replaced by a single period.



- :E** This replaces each word with its suffix (“extension”), so “\$(OBJS:E)” would give you “.OBJ .A”.
- :R** This replaces each word with everything but the suffix (thus returning the “root” of the word). “\$(OBJS:R)” expands to “..\LIB\A B \USR\LIB\LIBM”.

In addition, another style of substitution is also supported. This looks like:

```
$( VARIABLE:search-string=replacement )
```

It must be the last modifier in the chain. The search is anchored at the end of each word, so only suffixes or whole words may be replaced.

10.13

10.13.5 Advanced pmake Techniques

This section is devoted to those facilities in **pmake** that allow you to do a great deal in a makefile with very little work, as well as do some things you couldn't do in make without a great deal of work (and perhaps the use of other programs). The problem with these features is that they must be handled with care, or you will end up with a mess.

10.13.5.1 Search Paths

The **pmake** tool supports the dispersal of files into multiple directories by allowing you to specify places to look for sources with .PATH targets in the makefile. The directories you give as sources for these targets make up a “search path.” Only those files used exclusively as sources are actually sought on a search path, the assumption being that anything listed as a target in the makefile can be created by the makefile and thus should be in the current directory.

There are two types of search paths in **pmake**: one is used for all types of files (including included makefiles) and is specified with a plain .PATH target (e.g. “.PATH : RCS”), while the other is specific to a certain type of file, as indicated by the file's suffix. A specific search path is indicated by immediately following the .PATH with the suffix of the file. For instance

```
.PATH.H : \GEOSDEV\DEVEL\APPL\WORPRO \GEOSDEV\DEVEL
```



would tell **pmake** to look in the directories
\\GEOSDEV\\DEVEL\\APPL\\WORPRO and \\GEOSDEV\\DEVEL for any files
whose suffix is .H.

The current directory is always consulted first to see if a file exists. Only if it cannot be found are the directories in the specific search path, followed by those in the general search path, consulted.

10.13

A search path is also used when expanding wildcard characters. If the pattern has a recognizable suffix on it, the path for that suffix will be used for the expansion. Otherwise the default search path is employed.

When a file is found in some directory other than the current one, all local variables that would have contained the target's name (.ALLSRC and .IMPSRC) will instead contain the path to the file, as found by **pmake**. Thus if you have a file ..\\LIB\\MUMBLE.C and a makefile

```
.PATH.C : ..\\LIB
MUMBLE.EXE      : MUMBLE.C
$(CCOM) -o $(.TARGET) $(.ALLSRC)
```

the command executed to create MUMBLE.EXE would be
"bcc -o MUMBLE ..\\LIB\\MUMBLE.C"

If a file exists in two directories on the same search path, the file in the first directory on the path will be the one **pmake** uses. So if you have a large system spread over many directories, it would behoove you to follow a naming convention that avoids such conflicts.

Something you should know about the way search paths are implemented is that each directory is read, and its contents cached, exactly once—when it is first encountered—so any changes to the directories while **pmake** is running will not be noted when searching for implicit sources, nor will they be found when **pmake** attempts to discover when the file was last modified, unless the file was created in the current directory.



10.13.5.2 Conditional Statements

Like a C compiler, **pmake** allows you to configure the makefile using conditional statements. A conditional looks like this:

```
#if <Boolean expression>
<lines>
#elif <another Boolean expression>
<more lines>
#else
<still more lines>
#endif
```

10.13

They may be nested to a depth of 30 and may occur anywhere (except in a comment, of course). The “#” must be the very first character on the line.

Each Boolean expression is made up of terms that look like function calls, the standard C Boolean operators `&&`, `||`, and `!`, and the standard relational operators `==`, `!=`, `>`, `>=`, `<`, and `<=`, with `==` and `!=` being overloaded to allow string comparisons as well. The `&&` operator represents logical AND; `||` is logical OR and `!` is logical NOT. The arithmetic and string operators take precedence over all three of these operators, while NOT takes precedence over AND, which takes precedence over OR. This precedence may be overridden with parentheses, and an expression may be parenthesized to any level. Each Boolean term looks like a call on one of four functions:

make	The syntax is <code>make(<i>target</i>)</code> where <i>target</i> is a target in the makefile. This is <i>true</i> if the given target was specified on the command line or as the source for a <code>.MAIN</code> target (note that the sources for <code>.MAIN</code> are only used if no targets were given on the command line).
defined	The syntax is <code>defined(<i>variable</i>)</code> and is true if <i>variable</i> is defined. Certain variables are defined in the system makefile that identify the system on which pmake is being run.
exists	The syntax is <code>exists(<i>file</i>)</code> and is true if the file can be found on the global search path (i.e. that defined by <code>.PATH</code> targets, not by <code>.PATHsuffix</code> targets).
empty	This syntax is much like the others, except the string inside the parentheses is of the same form as you would put between parentheses when expanding a variable, complete with

modifiers. The function returns true if the resulting string is empty. (Note: an undefined variable in this context will cause at the very least a warning message about a malformed conditional, and at the worst will cause the process to stop once it has read the makefile. If you want to check for a variable being defined or empty, use the expression

```
!defined(var) || empty(var)
```

10.13

as the definition of `||` will prevent the `empty()` from being evaluated and causing an error, if the variable is undefined). This can be used to see if a variable contains a given word, for example:

```
#if !empty(var:Mword)
```

The arithmetic and string operators may only be used to test the value of a variable. The left-hand side must contain the variable expansion, while the right-hand side contains either a string, enclosed in double-quotes, or a number. The standard C numeric conventions (except for specifying an octal number) apply to both sides. For example,

```
#if $(OS) == 4.3
#if $(MACHINE) == "sun3"
#if $(LOAD_ADDR) < 0xc000
```

are all valid conditionals. In addition, the numeric value of a variable can be tested as a Boolean as follows:

```
#if $(LOAD)
```

would see if `LOAD` contains a non-zero value and

```
#if !$(LOAD)
```

would test if `LOAD` contains a zero value.

In addition to the bare `#if`, there are other forms that apply one of the first two functions to each term. They are as follows:

<code>ifdef</code>	<code>defined</code>
<code>ifndef</code>	<code>!defined</code>
<code>ifmake</code>	<code>make</code>
<code>ifnmake</code>	<code>!make</code>



There are also the “else if” forms: `elif`, `elifdef`, `elifndef`, `elifmake`, and `elifnmake`.

10.13.6 The Way Things Work

When **pmake** reads the makefile, it parses sources and targets into nodes in a graph. The graph is directed only in the sense that **pmake** knows which way is up. Each node contains not only links to all its parents and children (the nodes that depend on it and those on which it depends, respectively), but also a count of the number of its children that have already been processed.

10.13

The most important thing to know about how **pmake** uses this graph is that the traversal is breadth-first and occurs in two passes.

After **pmake** has parsed the makefile, it begins with the nodes the user has told it to make (either on the command line, or via a `.MAIN` target, or by the target being the first in the file not labeled with the `.NOTMAIN` attribute) placed in a queue. It continues to take the node off the front of the queue, mark it as something that needs to be made, pass the node to `Suff_FindDeps()` (mentioned earlier) to find any implicit sources for the node, and place all the node’s children that have yet to be marked at the end of the queue. If any of the children is a `.USE` rule, its attributes are applied to the parent, then its commands are appended to the parent’s list of commands and its children are linked to its parent. The parent’s `unmade-children` counter is then decremented (since the `.USE` node has been processed). This allows a `.USE` node to have children that are `.USE` nodes, and the rules will be applied in sequence. If the node has no children, it is placed at the end of another queue to be examined in the second pass. This process continues until the first queue is empty.

At this point, all the leaves of the graph are in the examination queue; **pmake** removes the node at the head of the queue and sees if it is out-of-date. If it is, it is passed to a function that will execute the commands for the node asynchronously. When the commands have completed, all the node’s parents have their `unmade-children` counter decremented and, if the counter is then zero, they are placed on the examination queue. Only those parents that were marked on the downward pass are processed in this way. Thus **pmake** traverses the graph back up to the nodes the user instructed it to create.

When the examination queue is empty and no shells are running to create a target, **pmake** is finished.

Once all targets have been processed, **pmake** executes the commands attached to the .END target, either explicitly or through the use of an ellipsis in a shell script. If there were no errors during the entire process but there are still some targets unmade (**pmake** keeps a running count of how many targets are left to be made), there is a cycle in the graph. The **pmake** program does a depth-first traversal of the graph to find all the targets that weren't made and prints them out one by one.

10.14

10.14 Swat Stub

The swat stub runs on the target machine, passing information between a running GEOS session and Swat on the host machine. It has one flag:

/H:hardware

Specify special hardware for sending signals. The only possible argument is **P** for sending via a serial port in a PCMCIA card.



Index



!	in makefiles TTools : 472, TTools : 495)	in Swat usage documentation TSwtA-I : 75
Tcl operator	TTCL : 273)	in Swat usage documentation TSwtA-I : 75
!=	TTools : 476	*	
in makefiles	TTools : 495		in makefile TTools : 474
Tcl operator	TTCL : 273		in Swat address expressions TSwatCm : 39
- (Hyphen)			in Swat usage documentation TSwtA-I : 76
Gcc command-line argument			Tcl operator TTCL : 273
TTools : 443		+	
in makefiles TTools : 475			in Swat address expressions TSwatCm : 38
#elif TTools : 495			in Swat usage documentation TSwtA-I : 76
#else TTools : 495			Tcl operator TTCL : 273
#endif TTools : 495		-	
#if TTools : 495			in Swat address expressions TSwatCm : 38
#ifdef TTools : 496			Tcl operator TTCL : 273
#ifmake TTools : 496		+=	TTools : 476
#ifndef TTools : 496		.	
#ifnmake TTools : 496			in Swat address expressions TSwatCm : 38
#include TTools : 486		.ALLSRC TTools : 478	
#undef TTools : 479		.asm files TTools : 436	
\$.BEGIN TTools : 489	
in Tcl TTCL : 270		.c files TTools : 435	
\$(VAR) expressions TTools : 477		.CFG file (SWAT.CFG file) TSwatCm : 35	
%		.def files TTools : 436	
Tcl operator TTCL : 273		.DEFAULT TTools : 489	
&		.dh files TTools : 435	
Tcl operator TTCL : 273		.doh files TTools : 435	
&&		.DONT CARE TTools : 487	
in makefiles TTools : 495		.ebj files TTools : 437	
Tcl operator TTCL : 273		.END TTools : 489	
Ctrl-]		.EXEC TTools : 487	
command completion in Swat		.geo files TTools : 437	
TSwatCm : 71		.goh files TTools : 435	
Ctrl-C		.gp files TTools : 436	
in Swat TSwatCm : 47		.h files TTools : 436	
Ctrl-D			
command completion in Swat			
TSwatCm : 71			
Escape key			
command completion in Swat			
TSwatCm : 71			
(



```

.IGNORE attribute  TTools : 487
.IGNORE target  TTools : 489
.IMP_SRC  TTools : 481
.INCLUDES  TTools : 489
.ldf files  TTools : 436
.LIBS  TTools : 489
.MAIN  TTools : 490
.MAKE  TTools : 487
.MAKEFLAGS global variable  TTools : 479
.MAKEFLAGS special target  TTools : 490
.mk files  TTools : 436
.NOTMAIN  TTools : 488
.obj files  TTools : 437
.OODATE  TTools : 478
.PATH  TTools : 490, TTools : 493
.ph files  TTools : 435
.PMAKE  TTools : 479
.poh files  TTools : 435
.PRECIOUS  TTools : 490
.PREFIX  TTools : 478
.RC file (SWAT.RC file)  TSwatCm : 72
.RECURSIVE  TTools : 490
.rev files  TTools : 436
.rsc files  TTools : 437
.SILENT  TTools : 488, TTools : 490
.SUFFIXES  TTools : 490
.sym files  TTools : 437
.TARGET  TTools : 478
.USE  TTools : 488
/
    Tcl operator  TTCL : 273
:
    changing threads in Swat
        TSwatCm : 68
    in makefiles  TTools : 472
    in Swat address expressions
        TSwatCm : 38
:: in makefiles  TTools : 472
:=  TTools : 476
:E  TTools : 493
:H  TTools : 492
:M  TTools : 491
:N  TTools : 491
:R  TTools : 493
:S  TTools : 492
:T  TTools : 492
:X  TTools : 491
<
    in makefiles  TTools : 495
    in Swat usage documentation
        TSwatA-I : 75
    Tcl operator  TTCL : 273
<<
    Tcl operator  TTCL : 273
<=
    in makefiles  TTools : 495

```

```

    Tcl operator  TTCL : 273
=  TTools : 476
    $(V:s=r) expressions  TTools : 493
==
    in makefiles  TTools : 495
    Tcl operator  TTCL : 273
>
    in makefiles  TTools : 495
    in Swat usage documentation
        TSwatA-I : 75
    Tcl operator  TTCL : 273
>=
    in makefiles  TTools : 495
    Tcl operator  TTCL : 273
>>
    Tcl operator  TTCL : 273
?
    in makefiles  TTools : 474
? =  TTools : 476
@
    Address history symbol in Swat
        TSwatCm : 70
    in makefiles  TTools : 475
[
    in makefiles  TTools : 474
    in Swat usage documentation
        TSwatA-I : 75
\ backslash substitution
    in Tcl  TTCL : 271
\\[*\\\\$(ROOT_DIR:T)\\\\\\\\*\\\\]\\\\\\\\*
    Example  TTools : 491
]
    in makefiles  TTools : 474
    in Swat usage documentation
        TSwatA-I : 75
Ctrl-]
    command completion in Swat
        TSwatCm : 71
^
    Command correction in Swat
        TSwatCm : 69
    Tcl operator  TTCL : 273
^h
    in Swat address expressions
        TSwatCm : 38
^l
    in Swat address expressions
        TSwatCm : 38
^v
    in Swat address expressions
        TSwatCm : 39
_print Swat command  TSwatA-I : 76
{ in makefiles  TTools : 473
|
    Tcl operator  TTCL : 273
||

```



in makefiles TTools : 495
 Tcl operator TTCL : 273
 } in makefiles TTools : 473
 ~
 Tcl operator TTCL : 273
 ... (Ellipsis) in makefiles TTools : 487
 ' (Backquote)
 In shell commands TTools : 476
 with != TTools : 476

A

abort TSwtA-I : 76
 abortframe TSwtA-I : 77
 Address expressions in Swat
 notation TSwatCm : 37
 of breakpoints TTCL : 306
 parsing TSwtA-I : 78, TSwtA-I : 79
 Address history in Swat TSwatCm : 70
 get-address Swat command
 TSwtA-I : 136
 addr-parse Swat command TSwtA-I : 78
 addr-preprocess Swat command
 TSwtA-I : 79
 addr-with-obj-flag Swat command
 TSwtA-I : 79
 alias Swat command TSwatCm : 65
 command reference TSwtA-I : 80
 alignFields Swat variable TSwtA-I : 82
 .ALLSRC TTools : 478
 antifreeze TSwtA-I : 82
 antithaw TSwtA-I : 83
 Applications, Sample
 ClipSamp TConfig : 23
 Controllers TConfig : 27, TConfig : 28
 CustGeom TConfig : 23
 Display Control TConfig : 25
 Document Control TConfig : 23
 Documents TConfig : 27
 Dynamic List TConfig : 25
 File Selector TConfig : 24
 File Selector Filters TConfig : 25
 GenAttrs TConfig : 25
 GenDynamicList TConfig : 27
 Generic Trees TConfig : 25
 GenGlyph TConfig : 25

GenInteraction TConfig : 25
 GenItemGroup TConfig : 25
 GenText TConfig : 27
 GenTrigger TConfig : 27
 GenView TConfig : 28
 Graphic Object TConfig : 25
 GStrings TConfig : 25
 Hello World TConfig : 25
 Input TConfig : 24
 Multiple Primaries TConfig : 27
 TicTac TConfig : 27
 Vardata TConfig : 28
 Visible Objects TConfig : 28
 VisMonikers TConfig : 26
 appobj Swat command TSwtA-I : 83
 apropos Swat command TSwatCm : 40,
 TSwtA-I : 84
 Arrays in Swat
 creating TTCL : 326
 examining TTCL : 327
 in example TTCL : 329
 setting elements TSwtA-I : 84
 aset Swat command TSwtA-I : 84
 .asm files TTools : 436
 ASMFLAGS variable TTools : 467
 assign Swat command TSwatCm : 64
 command reference TSwtA-I : 85
 assoc Swat command TSwtA-I : 85
 Associative lists in Swat
 deleting elements TSwtA-I : 105
 searching TSwtA-I : 85
 att Swat command TSwatCm : 45
 command reference TSwtA-I : 86
 attach Swat command TSwatCm : 45
 command reference TSwtA-I : 86
 autoload Swat command TTCL : 331
 command reference TSwtA-I : 87

B

Ctrl-b Swat navigation TSwatCm : 69
 \ backslash substitution
 in Tcl TTCL : 271
 backtrace Swat command TSwatCm : 56
 command reference TSwtA-I : 88
 bc TTCL : 278



.BEGIN TTools : 489
 bindings Swat command TSwtA-I : 89
 bind-key Swat command TSwtA-I : 89
 Bitmaps
 examining with Swat TSwtJ-Z : 178,
 TSwtJ-Z : 180, TSwtJ-Z : 191
 break Tcl command TTCL : 279
 Breakpoints TSwtCm : 47–TSwtCm : 51,
 TTCL : 303
 "hardware breakpoints" TSwtA-I : 142
 conditional TSwtCm : 50, TTCL : 304,
 TTCL : 308
 detecting TSwtA-I : 89
 on handles TSwtA-I : 90
 setting TSwtJ-Z : 233
 setting interactively TSwtA-I : 149
 single use TSwtA-I : 138
 tally breakpoints TSwtJ-Z : 243
 timing breakpoints TSwtJ-Z : 247
 break-taken Swat command TSwtA-I : 89
 brk Tcl structure TSwtCm : 49
 command reference TTCL : 303
 brkload Swat command TSwtA-I : 90
 byteAsChar Swat variable TSwtA-I : 90
 ByteFlags
 printing flag fields TSwtJ-Z : 195
 bytes Swat command TSwtCm : 55
 command reference TSwtA-I : 91
 Tcl source TTCL : 329

C

Ctrl-C
 in Swat TSwtCm : 47
 .c files TTools : 435
 cache Tcl structure TTCL : 306
 call Swat command TSwtA-I : 91
 call-patient Swat command TSwtA-I : 92
 car Swat command TSwtA-I : 93
 case Tcl command TTCL : 279
 catch Tcl command TTCL : 281
 cbrk Swat command TSwtCm : 50
 cbrk Tcl structure
 command reference TTCL : 308
 CCOM variable TTools : 468
 CCOMFLAGS variable TTools : 467

cdr Swat command TSwtA-I : 93
 Cells
 examining with Swat TSwtJ-Z : 198
 printing cell data with Swat
 TSwtJ-Z : 181
 printing cell dependencies with Swat
 TSwtJ-Z : 181
 printing ColumnArrayElement
 structures in Swat
 TSwtJ-Z : 199
 printing dependency lists with Swat
 TSwtJ-Z : 200
 printing paramaters in Swat
 TSwtJ-Z : 198
 printing row blocks in Swat
 TSwtJ-Z : 202
 printing spreadheets in Swat
 TSwtJ-Z : 205
 .CFG file (SWAT.CFG file) TSwtCm : 35
 CFLAGS TTools : 481
 Chunk arrays
 examining with Swat TSwtJ-Z : 179
 classes Swat command TSwtA-I : 93
 Clipboard
 print-clipboard-item Swat command
 TSwtJ-Z : 199
 printing transfer item with Swat
 TSwtJ-Z : 191
 clrc Swat command TSwtA-I : 94
 columns Swat command TSwtA-I : 94
 Command completion in Swat
 TSwtCm : 71
 abbreviations TSwtA-I : 80
 completion Swat command TSwtA-I : 95
 Command correction in Swat TSwtCm : 69
 COMMAND.COM TTools : 476
 Command-line Variables
 TTools : 478–TTools : 479
 Comments
 In makefiles TTools : 480
 compcc Swat command TSwtA-I : 95
 completion Swat command TSwtA-I : 95
 concat Tcl command TTCL : 282
 condenseSmall Swat variable TSwtA-I : 96
 condenseSpecial Swat variable TSwtA-I : 96
 Conditional statements in makefiles
 TTools : 495
 cont Swat command TSwtCm : 46



- command reference TSwtA-I : 96
- Content
 - finding with Swat TSwtA-I : 80, TSwtA-I : 97
- content Swat command TSwtA-I : 97
- continue Tcl command TTCL : 282
- continue-patient Swat command TSwtA-I : 97
- CPP variable TTools : 467
- Ctrl-]
 - command completion in Swat TSwtCm : 71
- Ctrl-b Swat navigation TSwtCm : 69
- Ctrl-C
 - in Swat TSwtCm : 47
- Ctrl-D
 - command completion in Swat TSwtCm : 71
- Ctrl-d Swat navigation TSwtCm : 69
- Ctrl-e Swat navigation TSwtCm : 69
- Ctrl-f Swat navigation TSwtCm : 69
- Ctrl-u Swat navigation TSwtCm : 69
- Ctrl-y Swat navigation TSwtCm : 69
- cup Swat command TSwtA-I : 98
- current-level Swat command TSwtA-I : 98
- current-registers Swat command TSwtA-I : 99
- cvtrecord Swat command TSwtA-I : 99
- cycles Swat command TSwtA-I : 100

D

- Ctrl-D
 - command completion in Swat TSwtCm : 71
- Ctrl-d Swat navigation TSwtCm : 69
- DBase
 - examining blocks with Swat TSwtJ-Z : 182
 - examining items with Swat TSwtJ-Z : 200
 - printing group blocks in Swat TSwtJ-Z : 199
- dcache Swat command TSwtA-I : 100
- dcall Swat command TSwtA-I : 101
- debug Swat command TSwtA-I : 102

- debugger Swat command TSwtA-I : 103
- debugOnError Swat variable TSwtA-I : 103
- .def files TTools : 436
- .DEFAULT TTools : 489
- defcmd Swat command TSwtA-I : 103
- defcommand Swat command TSwtA-I : 104
 - in example TTCL : 328
- defhelp Swat command TSwtA-I : 104
- defined() conditional TTools : 495
- defsubr Tcl command TTCL : 283
- defvar Swat command TSwtA-I : 105
- delassoc Swat command TSwtA-I : 105
- Depend target TTools : 466
- Dependency lines TTools : 471
- detach Swat command TSwtCm : 46
 - command reference TSwtA-I : 106
- .dh files TTools : 435
- Directories, Host TConfig : 15
- Directories, Target TConfig : 19
- dirs Swat command TSwtA-I : 106
- discard-state Swat command TSwtA-I : 107
- diskwalk Swat command TSwtA-I : 107
- display Swat command TSwtA-I : 108
- Display windows in Swat TSwtA-I : 108
 - clearing TSwtJ-Z : 257
 - columns Swat command TSwtA-I : 94
 - creating TSwtJ-Z : 257
 - destroying TSwtJ-Z : 257
 - displaying current Stack frame TSwtA-I : 130
 - displaying registers TSwtJ-Z : 215
 - displaying source code TSwtJ-Z : 229
 - displaying variable TSwtJ-Z : 253
 - flags register TSwtA-I : 123
 - redisplaying TSwtJ-Z : 261
 - wpop Swat command TSwtJ-Z : 260
 - wpush Swat command TSwtJ-Z : 260
- doc Swat command TSwtCm : 40
 - reference TSwtA-I : 108
- doc-next Swat command TSwtA-I : 109
- doc-previous Swat command TSwtA-I : 109
- .doh files TTools : 435
- Dollar sign
 - in Tcl TTCL : 270
- .DONTCARE TTools : 487
- dosMem Swat command TSwtA-I : 109
- down Swat command TSwtA-I : 110
- drivewalk Swat command TSwtA-I : 110



Dumping Swat output TSwtJ-Z : 219
 dumpstack Swat command TSwtA-I : 111
 dwordIsPtr Swat variable TSwtA-I : 112
 dwords Swat command TSwtCm : 55
 command reference TSwtA-I : 112

E

:E TTools : 493
 Ctrl-e Swat navigation TSwtCm : 69
 .ebj files TTools : 437
 EC files TTools : 437
 ec Swat command TSwtA-I : 113
 ec.geo files TTools : 437
 ec.sym files TTools : 437
 echo Swat command TSwtA-I : 114
 in example TTCL : 329,
 TTCL : 329–TTCL : 330
 #elif TTools : 495
 elist Swat command TSwtA-I : 115
 #else TTools : 495
 empty() conditional TTools : 495
 .END TTools : 489
 #endif TTools : 495
 ensure-swat-attached Swat command
 TSwtA-I : 115
 Environment variables
 accessing from Swat TSwtA-I : 137
 Environment variables within PMake
 TTools : 479
 eqfind Swat command TSwtA-I : 115
 eqlist Swat command TSwtA-I : 116
 erfind Swat command TSwtA-I : 116
 error Tcl command TTCL : 283
 Errors
 continuing after in Swat TSwtCm : 46,
 TSwtA-I : 96
 continuing after with Swat TSwtA-I : 97
 continuing after, in Swat TSwtA-I : 149
 decoding FatalErrors values
 TSwtJ-Z : 183, TSwtJ-Z : 259
 determining cause of crash
 TSwtA-I : 118, TSwtJ-Z : 259
 error checking level TSwtA-I : 113
 fatalerr_auto_explain Swat variable
 TSwtA-I : 119
 Swat timeout errors TSwtCm : 35
 Tcl errors TTCL : 276
 Escape key
 command completion in Swat
 TSwtCm : 71
 eval Tcl command TTCL : 284
 event Tcl structure TTCL : 309
 Events
 displaying TSwtA-I : 116
 displaying with Swat TSwtA-I : 115
 finding recorded TSwtA-I : 116
 handles TTCL : 313
 printing with Swat TSwtJ-Z : 183
 .EXEC TTools : 487
 exists()conditional TTools : 495
 exit TSwtA-I : 117
 exit-thread Swat command TSwtA-I : 117
 explain Swat command TSwtA-I : 118
 explode Swat command TSwtA-I : 118
 expr Tcl command TTCL : 284
 in example TTCL : 329–TTCL : 330

F

Ctrl-f Swat navigation TSwtCm : 69
 fatalerr_auto_explain Swat variable
 TSwtA-I : 119
 fetch-optr Swat command TSwtA-I : 119
 fhandle Swat command TSwtA-I : 120
 field Swat command TSwtA-I : 120
 fieldwin Swat command TSwtA-I : 121
 file Tcl command TTCL : 285
 Files
 Application Files TConfig : 17
 displaying open files TSwtA-I : 133
 DOS files TSwtJ-Z : 241
 Driver Files TConfig : 15
 examining directory stack with Swat
 TSwtA-I : 106
 examining standard paths with Swat
 TSwtJ-Z : 231
 fhandle Swat command TSwtA-I : 120
 handles TTCL : 313
 Included Files TConfig : 16
 Library Files TConfig : 15



- listing current disks with Swat
 - TSwtA-I : 107
- listing Drives with Swat TSwtA-I : 110
- listing with Swat TSwtA-I : 133,
 - TSwtA-I : 135, TSwtJ-Z : 224,
 - TSwtJ-Z : 241
- Loading Tcl files TSwtJ-Z : 162
- loading Tcl files TSwtA-I : 87,
 - TTCL : 330
- manipulating host machine files with
 - Swat TSwtJ-Z : 235
- printing current directory in Swat
 - TSwtJ-Z : 213
- printing disk information with Swat
 - TSwtJ-Z : 182
- Sample Application Files TConfig : 18
- TCL Files TConfig : 18
- Files, Tool Files TConfig : 18
- find Swat command TSwtA-I : 121
- find-opcode Swat command TSwtA-I : 121
- finish Swat command TSwtA-I : 122
- finishframe Swat command TSwtA-I : 123
- Flags
 - examining flags fields TSwtJ-Z : 195
- Flags register
 - clearing with Swat TSwtA-I : 94
 - complementing flags with Swat
 - TSwtA-I : 95
 - getting flags with Swat TSwtA-I : 136
 - in display window TSwtA-I : 123
 - printing TSwtJ-Z : 183
 - setting flags with Swat TSwtJ-Z : 223
- flagwin Swat command TSwtA-I : 123
- flowobj Swat command TSwtA-I : 123
- flush-output Swat command TSwtA-I : 124
- fntoptr Swat command TSwtA-I : 124
- fntval Swat command TSwtA-I : 125
- Focus TSwtA-I : 80
 - finding with Swat TSwtA-I : 125,
 - TSwtA-I : 126
- focus Swat command TSwtA-I : 125
- focusobj Swat command TSwtA-I : 126
- Fonts
 - examining with Swat TSwtJ-Z : 184
 - monitoring with Swat TSwtJ-Z : 208
- fonts Swat command TSwtA-I : 126
- for Tcl command TTCL : 286
 - in example TTCL : 329

- foreach Tcl command TTCL : 287
- format Tcl command TTCL : 288
 - in example TTCL : 330
- Formatting Swat output TSwtCm : 58,
 - TSwtA-I : 76, TSwtA-I : 114,
 - TSwtA-I : 125
- formatting numbers TSwtA-I : 145
- integer values of expressions
 - TSwtA-I : 137
- inverse type TSwtJ-Z : 259
- moving cursor TSwtJ-Z : 260
- fpstack Swat command TSwtA-I : 127
- fpu-state Swat command TSwtA-I : 127
- frame Swat command TSwtCm : 56
 - command reference TSwtA-I : 128
- framewin Swat command TSwtA-I : 130
- freeze Swat command TSwtA-I : 131
- fullscreen TSwtA-I : 131
- func Swat command TSwtA-I : 132
- fvardata Swat command TSwtA-I : 132
- fwalk Swat command TSwtA-I : 133

G

- gc Swat command TSwtA-I : 133
- GenApplication object TSwtA-I : 80,
 - TSwtA-I : 83
- Generic objects
 - accessing parents TSwtCm : 63
 - displaying ancestors TSwtA-I : 139
 - displaying instance data with Swat
 - TSwtJ-Z : 185
 - displaying tree TSwtCm : 62,
 - TSwtA-I : 134
- gentree Swat command TSwtCm : 62
 - command reference TSwtA-I : 134
- .geo files TTools : 437
- Geode files TTools : 437
- Geode parameters files TTools : 436
- GEODE variable TTools : 467
- Geodes
 - displaying TSwtA-I : 145
 - loading with Swat TSwtJ-Z : 162
- GEOS.INI TIni : 363, TIni : 364
- GEOS.INI file
 - examining with Swat TSwtJ-Z : 188



geosfiles Swat command TSwtA-I : 135
 geos-release Swat command TSwtA-I : 135
 geowatch Swat command TSwtA-I : 135
 get-address Swat command TSwtA-I : 136
 in example TTCL : 329
 getcc Swat command TSwtA-I : 136
 getenv Swat command TSwtA-I : 137
 get-key-binding Swat command
 TSwtA-I : 137
 getvalue Swat command TSwtA-I : 137
 global Tcl command TTCL : 288
 Global variables TTools : 479
 Glue TTools : 440
 Glue parameters files TTools : 436
 go Swat command TSwtA-I : 138
 Goc TTools : 443–TTools : 444
 GOCFLAGS variable TTools : 467
 .goh files TTools : 435
 .gp files TTools : 436
 Grev TTools : 444–TTools : 447
 grobjtree Swat command TSwtA-I : 138
 GStrings
 examining with Swat TSwtJ-Z : 185
 printing elements stored in a chunk array
 TSwtJ-Z : 180
 gup Swat command TSwtCm : 63
 command reference TSwtA-I : 139

H

:H TTools : 492
 ^h
 in Swat address expressions
 TSwtCm : 38
 .h files TTools : 436
 handle Tcl structure TTCL : 310
 Handles
 dereferencing in Swat expressions
 TSwtCm : 38
 displaying information with Swat
 TSwtJ-Z : 204
 displaying the handle table
 TSwtCm : 58
 displaying with Swat TSwtA-I : 141
 examining with Swat TSwtJ-Z : 186
 file handles

fhandles Swat command
 TSwtA-I : 120
 in breakpoint locations TTCL : 305
 monitoring with Swat TSwtA-I : 90
 handles Swat command TSwtA-I : 139
 handsum Swat command TSwtA-I : 141
 hbrk Swat command TSwtA-I : 142
 Heap
 error checking TSwtA-I : 114
 examining TSwtA-I : 146
 examining with Swat TSwtCm : 58
 heap space Swat command
 TSwtA-I : 142
 monitoring with Swat TSwtA-I : 142
 heap space Swat command TSwtA-I : 142
 help Swat command
 TSwtCm : 40–TSwtCm : 41
 command reference TSwtA-I : 143
 defining help for commands
 TSwtA-I : 103, TSwtA-I : 104
 defining help for variables
 TSwtA-I : 105
 defining new help topics TSwtA-I : 104
 help-fetch Swat command TSwtA-I : 143
 help-fetch-level Swat command
 TSwtA-I : 143
 help-help Swat command TSwtA-I : 144
 help-is-leaf Swat command TSwtA-I : 144
 help-minAspect Swat command
 TSwtA-I : 144
 help-scan Swat command TSwtA-I : 145
 help-verbose Swat variable TSwtA-I : 145
 hex Swat command TSwtA-I : 145
 hgwalk Swat command TSwtA-I : 145
 History
 Address, in Swat TSwtCm : 70
 Command, in Swat TSwtCm : 69
 history Swat command TSwtA-I : 146
 most recently accessed address
 TSwtA-I : 136
 history Swat command TSwtA-I : 146
 Huge arrays
 printing with Swat TSwtJ-Z : 187
 hwalk Swat command TSwtCm : 58
 command reference TSwtA-I : 146



I

iacp Swat command TSwtA-I : 148
 ibrk Swat command TSwtA-I : 149
 ibrkPageLen Swat variable TSwtA-I : 149
 #if TTools : 495
 if Tcl command TTCL : 289
 in example TTCL : 329
 #ifdef TTools : 496
 #ifmake TTools : 496
 #ifndef TTools : 496
 #ifnmake TTools : 496
 ignerr Swat command TSwtA-I : 149
 .IGNORE attribute TTools : 487
 .IGNORE target TTools : 489
 imem Swat command TSwtCm : 64
 command reference TSwtA-I : 150
 imemPageLen Swat variable TSwtA-I : 152
 implied source TTools : 481
 impliedgrab Swat command TSwtA-I : 152
 .IMPSRC TTools : 481
 .INCLUDES TTools : 489
 index Tcl command TTCL : 290
 in example TTCL : 329
 info Tcl command TTCL : 291
 INI File TIni : 363
 Categories TIni : 364
 Initialization file for Swat TSwtCm : 72
 Instance data
 displaying GenClass TSwtJ-Z : 185
 displaying VisClass TSwtJ-Z : 209
 examining with Swat TSwtCm : 60,
 TSwtJ-Z : 188, TSwtJ-Z : 189
 int Swat command TSwtA-I : 153
 .INTERRUPT TTools : 489
 Interrupts TSwtA-I : 153
 manipulating with Swat TSwtA-I : 154,
 TSwtA-I : 155
 masking with Swat TSwtJ-Z : 223
 monitoring with Swat TSwtA-I : 153
 intFormat Swat variable TSwtA-I : 154
 intr Swat command TSwtA-I : 154
 I/O manipulating with Swat TSwtA-I : 154
 io Swat command TSwtA-I : 154
 irq Swat command TSwtA-I : 155
 is-obj-in-class Swat command TSwtA-I : 156
 istep Swat command TSwtCm : 52

command reference TSwtA-I : 156

K

Key bindings in Swat TSwtA-I : 89,
 TSwtA-I : 137
 removing TSwtJ-Z : 250
 Keyboard
 allowing interrupts TSwtA-I : 153
 displaying keyboard hierarchy with Swat
 TSwtJ-Z : 159
 finding keyboard grab with Swat
 TSwtJ-Z : 159
 focus TSwtA-I : 80
 grab TSwtA-I : 80
 keyboard Swat command TSwtJ-Z : 159
 keyboardobj Swat command TSwtJ-Z : 159

L

lastCommand Swat command TSwtJ-Z : 160
 .ldf files TTools : 436
 length Tcl command TTCL : 292
 lhwalk Swat command TSwtCm : 59
 command reference TSwtJ-Z : 160
 Lib target TTools : 466
 Library definition files TTools : 436
 .LIBS TTools : 489
 link Swat command TSwtJ-Z : 160
 LINKFLAGS variable TTools : 467
 list Tcl command TTCL : 293
 listi Swat command
 command reference TSwtJ-Z : 161
 Lists in Swat
 car Swat command TSwtA-I : 93
 cdr Swat command TSwtA-I : 93
 concat Tcl command TTCL : 282
 length Tcl command TTCL : 292
 list Tcl command TTCL : 293
 range Tcl command TTCL : 295
 Tcl syntax TTCL : 275
 load TTCL : 331
 load Swat command
 command reference TSwtJ-Z : 162
 loadapp Swat command TSwtJ-Z : 162



loadgeode Swat command TSwtJ-Z : 162
 Local memory
 address expressions in Swat
 TSwatCm : 38
 detecting heaps TSwtA-I : 140,
 TSwtA-I : 148
 displaying TSwatCm : 59
 error checking TSwtA-I : 114
 examaning with Swat TSwtJ-Z : 160
 examining chunk arrays with Swat
 TSwtJ-Z : 179
 examining with Swat TSwtJ-Z : 190
 handles TTCL : 313
 monitoring with Swat TSwtJ-Z : 226
 Local variables TTools : 478
 displaying with Swat TSwtJ-Z : 163
 Local.mk TTools : 467
 locals Swat command TSwtJ-Z : 163
 localwin Swat command TSwtJ-Z : 163
 loop Swat command TSwtJ-Z : 163

M

:M TTools : 491
 Machine Setup TConfig : 21
 .MAIN TTools : 490
 Main target TTools : 490
 .MAKE TTools : 487
 make() conditional TTools : 495
 Makefiles TTools : 436
 .MAKEFLAGS global variable TTools : 479
 .MAKEFLAGS special target TTools : 490
 map Swat command TSwtJ-Z : 164
 in example TTCL : 330
 mapconcat Swat command TSwtJ-Z : 164
 in example TTCL : 330
 map-method Swat command TSwtJ-Z : 165
 mcount Swat command TSwtJ-Z : 166
 Memory
 displaying usage with Swat
 TSwtA-I : 145
 examining DOS memory with Swat
 TSwtA-I : 109
 examining with Swat TSwtA-I : 91,
 TSwtA-I : 112, TSwtA-I : 150,
 TSwtJ-Z : 260

finding size with Swat TSwtJ-Z : 166
 monitoring checksum with Swat
 TSwtA-I : 113
 monitoring with Swat TSwtA-I : 142,
 TSwtJ-Z : 248
 Memory modes TSwtA-I : 150
 memsize Swat command TSwtJ-Z : 166
 Messages
 decoding message values TSwtJ-Z : 165
 monitoring with Swat TSwatCm : 66,
 TSwtJ-Z : 166, TSwtJ-Z : 169,
 TSwtJ-Z : 171, TSwtJ-Z : 174,
 TSwtJ-Z : 175, TSwtJ-Z : 203,
 TSwtJ-Z : 225, TSwtJ-Z : 226
 sending with Swat TSwtJ-Z : 175
 verifying receipt TSwatCm : 67
 Methods
 finishing stepping through
 TSwatCm : 53
 stopping in with Swat TSwatCm : 53,
 TSwtJ-Z : 231
 methods TSwtJ-Z : 167
 .mk files TTools : 436
 Mkmf TTools : 447–TTools : 448
 Model
 finding with Swat TSwtA-I : 80,
 TSwtJ-Z : 167, TSwtJ-Z : 168
 model Swat command TSwtJ-Z : 167
 modelobj Swat command TSwtJ-Z : 168
 Monikers
 displaying with Swat TSwtJ-Z : 185
 examining with Swat TSwtJ-Z : 194,
 TSwtJ-Z : 210
 Mouse
 finding mouse grab with Swat
 TSwtA-I : 80, TSwtJ-Z : 169
 printing mouse hierarchy TSwtJ-Z : 168
 Mouse pointer
 object under TSwtA-I : 152
 Swat usage TSwatCm : 69
 window under TSwtA-I : 80
 mouse Swat command TSwtJ-Z : 168
 mouseobj Swat command TSwtJ-Z : 169
 mwatch Swat command TSwatCm : 66,
 TSwtJ-Z : 169



N

:N TTools : 491
 next Swat command TSwtJ-Z : 170
 NO_EC variable TTools : 467
 noStructEnum Swat variable TSwtJ-Z : 170
 .NOTMAIN TTools : 488
 null Swat command TSwtJ-Z : 171
 in example TTCL : 329

O

.obj files TTools : 437
 objbrk Swat command TSwtJ-Z : 171
 obj-class Swat command TSwtJ-Z : 172
 objcount Swat command TSwtJ-Z : 172
 Object files TTools : 437
 Objects
 application object TSwtA-I : 80,
 TSwtA-I : 83
 as Tcl symbols TTCL : 319
 displaying generic instance data
 TSwtJ-Z : 185
 displaying object blocks TSwtCm : 59
 displaying text objects with Swat
 TSwtJ-Z : 206
 displaying variable data with Swat
 TSwtJ-Z : 209
 examining instance data TSwtCm : 60
 examining monikers with Swat
 TSwtJ-Z : 210
 examining with Swat TSwtJ-Z : 174,
 TSwtJ-Z : 188, TSwtJ-Z : 189,
 TSwtJ-Z : 192, TSwtJ-Z : 194
 finding variable data TSwtA-I : 132
 lmem object error checking
 TSwtA-I : 114
 monitoring process object TSwtJ-Z : 203
 monitoring with Swat TSwtCm : 67,
 TSwtA-I : 135, TSwtJ-Z : 171,
 TSwtJ-Z : 172, TSwtJ-Z : 174,
 TSwtJ-Z : 175
 printing class TSwtJ-Z : 181
 printing class hierarchy in Swat
 TSwtJ-Z : 206

 printing visual bounds with Swat
 TSwtJ-Z : 212
 testing class with Swat TSwtA-I : 156
 obj-foreach-class Swat command
 TSwtJ-Z : 173
 objmessagebrk Swat command
 TSwtJ-Z : 174
 objwalk Swat command TSwtCm : 59,
 TSwtJ-Z : 174
 objwatch Swat command TSwtCm : 67,
 TSwtJ-Z : 175
 omfq Swat command TSwtJ-Z : 175
 .OODATE TTools : 478
 Optimization
 tally breakpoints TSwtJ-Z : 243
 optr
 dereferencing in Swat address
 expressions TSwtCm : 38
 from global and local handle
 TSwtA-I : 124
 returning Tcl type token for TTCL : 324
 optrs
 extracting from memory with Swat
 TSwtA-I : 119

P

pappcache TSwtJ-Z : 176
 patch Swat command TSwtJ-Z : 176
 patchin Swat command TSwtJ-Z : 177
 patchout Swat command TSwtJ-Z : 178
 .PATH TTools : 490, TTools : 493
 PATH additions TConfig : 21
 Paths, graphics
 displaying with Swat TSwtJ-Z : 194
 patient Tcl structure TTCL : 314
 Patients
 application object TSwtA-I : 80,
 TSwtA-I : 83
 displaying handles TSwtA-I : 140
 displaying status with Swat
 TSwtJ-Z : 204
 listing active frames of TSwtA-I : 88
 name in symbols TTCL : 319
 patient Tcl structure TTCL : 314
 setting the default TSwtCm : 68



specific breakpoints TTCL : 303
 switching TSwtJ-Z : 239
 switching between TSwatCm : 68
 waking up with Swat TSwtJ-Z : 257
 pbitmap Swat command TSwtJ-Z : 178
 pbody Swat command TSwtJ-Z : 178
 parray Swat command TSwtJ-Z : 179
 pbitmap Swat command TSwtJ-Z : 180
 Pccom TTools : 448
 pcelldata Swat command TSwtJ-Z : 181
 pcelldeps Swat command TSwtJ-Z : 181
 Pcget TTools : 461
 pclass Swat command TSwtJ-Z : 181
 Pcs TTools : 462–TTools : 464
 PCS.PAT file TTools : 462
 Pcsend TTools : 465
 pdb Swat command TSwtJ-Z : 182
 pdisk Swat command TSwtJ-Z : 182
 pdrive TSwtJ-Z : 182
 penum Swat command TSwtJ-Z : 183
 pevent Swat command TSwtJ-Z : 183
 pflags Swat command TSwtJ-Z : 183
 pfont Swat command TSwtJ-Z : 184
 pfontinfo Swat command TSwtJ-Z : 184
 pgen Swat command TSwtJ-Z : 185
 pgs Swat command TSwtJ-Z : 185
 .ph files TTools : 435
 phandle Swat command TSwtJ-Z : 186
 pharray Swat command TSwtJ-Z : 187
 pini Swat command TSwtJ-Z : 188
 piv Swat command TSwtJ-Z : 189
 plines Swat command TSwtJ-Z : 190
 plist Swat command TSwtJ-Z : 190
 .PMAKE TTools : 479
 PMAKE TTools : 479
 PMake TTools : 465–TTools : 498
 pncbitmap Swat command TSwtJ-Z : 191
 pnormal Swat command TSwtJ-Z : 191
 pobjarray Swat command TSwtJ-Z : 192
 pobject Swat command TSwatCm : 60
 command reference TSwtJ-Z : 192
 pobjmon Swat command TSwtJ-Z : 194
 pod Swat command TSwtJ-Z : 194
 .poh files TTools : 435
 Pointers
 dereferencing in Swat address
 expressions TSwatCm : 39
 segment/offset pairs in Swat address
 expressions TSwatCm : 38
 ppath Swat command TSwtJ-Z : 194
 .PRECIOUS TTools : 488, TTools : 490
 precord Swat command TSwtJ-Z : 195
 .PREFIX TTools : 478
 preg Swat command TSwtJ-Z : 196
 print Swat command TSwatCm : 58
 command reference TSwtJ-Z : 196
 print-cell Swat command TSwtJ-Z : 198
 print-cell-params Swat command
 TSwtJ-Z : 198
 print-clipboard-item Swat command
 TSwtJ-Z : 199
 print-column-element Swat command
 TSwtJ-Z : 199
 print-db-group Swat command
 TSwtJ-Z : 199
 print-db-item Swat command TSwtJ-Z : 200
 print-eval-dep-list Swat command
 TSwtJ-Z : 200
 printNamesInObjTrees Swat variable
 TSwtJ-Z : 200
 print-obj-and-method Swat command
 TSwtJ-Z : 201
 printRegions Swat variable TSwtJ-Z : 201
 print-row Swat command TSwtJ-Z : 202
 print-row-block Swat command
 TSwtJ-Z : 202
 printStop Swat command TSwtJ-Z : 203
 proc Tcl command TTCL : 293
 procmessagebrk Swat command
 TSwtJ-Z : 203
 protect Tcl command TTCL : 295
 ps Swat command TSwtJ-Z : 204
 pscope Swat command TSwtJ-Z : 204
 psize Swat command TSwtJ-Z : 205
 pssheet Swat command TSwtJ-Z : 205
 psup Swat command TSwtJ-Z : 206
 ptext Swat command TSwtJ-Z : 206
 pthread Swat command TSwtJ-Z : 207
 ptimer Swat command TSwtJ-Z : 207
 ptrans Swat command TSwtJ-Z : 207
 ptreg Swat command TSwtJ-Z : 208
 PTTY TConfig : 22
 On target machine TConfig : 22
 pusage Swat command TSwtJ-Z : 208
 pvardata Swat command TSwtJ-Z : 209



pvardentry Swat command TSwtJ-Z : 209
 pvis Swat command TSwtJ-Z : 209
 pvismon Swat command TSwtJ-Z : 210
 pvmb Swat command TSwtJ-Z : 210
 pvmt Swat command TSwtJ-Z : 211
 pvsize Swat command TSwtJ-Z : 212
 pwd Swat command TSwtJ-Z : 213

Q

Queues, event
 displaying events in TSwtA-I : 116
 finding with Swat TSwtA-I : 115
 handles TTCL : 313
 in optrs TSwtA-I : 124
 quit Swat command TSwtCm : 46
 command reference TSwtJ-Z : 213

R

:R TTools : 493
 range Tcl command TTCL : 295
 in example TTCL : 329
 .RC file (SWAT.RC file) TSwtCm : 72
 read-char Swat command TSwtJ-Z : 213
 read-line Swat command TSwtJ-Z : 214
 read-reg Swat command TSwtJ-Z : 215
 .RECURSIVE TTools : 490
 Regions, graphics
 examining with Swat TSwtJ-Z : 196
 printing in Swat TSwtJ-Z : 208
 printRegions Swat variable
 TSwtJ-Z : 201
 Registers
 accessing with Swat TSwtJ-Z : 215
 current-registers Swat command
 TSwtA-I : 99
 displaying with Swat TSwtJ-Z : 215
 error checking TSwtA-I : 114
 flag register
 clearing flags with Swat
 TSwtA-I : 94
 flags register
 complementing flags with Swat
 TSwtA-I : 95

getting flags with Swat
 TSwtA-I : 136
 printing TSwtJ-Z : 183
 setting flags with Swat
 TSwtJ-Z : 223

flags register
 displaying TSwtA-I : 123
 monitoring with Swat TSwtJ-Z : 215
 setting value with Swat TSwtA-I : 85
 value within a frame TSwtA-I : 128
 value within thread TTCL : 322
 regs Swat command TSwtJ-Z : 215
 regwin Swat command TSwtJ-Z : 215
 repeatCommand Swat variable
 TSwtJ-Z : 216
 require Swat command TSwtJ-Z : 216
 restore-state Swat command TSwtJ-Z : 217
 ret Swat command TSwtJ-Z : 217
 return Tcl command TTCL : 296
 return-to-top-level Swat command
 TSwtJ-Z : 218
 .rev files TTools : 436
 Revision files TTools : 436
 ROOT_DIR TConfig : 22
 Routines
 calling from Swat TSwtA-I : 91,
 TSwtA-I : 92
 current routine TSwtA-I : 132
 current within frame TSwtA-I : 130
 local variables TSwtJ-Z : 163
 monitoring with Swat TSwtA-I : 101
 rs Swat command TSwtJ-Z : 218
 .rsc files TTools : 437
 run TSwtJ-Z : 218
 rwatch Swat command TSwtJ-Z : 219

S

:S TTools : 492
 save Swat command TSwtJ-Z : 219
 save-state Swat command TSwtJ-Z : 220
 sbwalk Swat command TSwtJ-Z : 220
 scan Tcl command TTCL : 297
 scope Swat command TSwtJ-Z : 221
 screenwin Swat command TSwtJ-Z : 221



- Search paths in makefiles TTools : 493
- Searching for text with Swat TSwtA-I : 121
- SEND file TTools : 462
- send Swat command
 - reference TSwtJ-Z : 222
- send-file TSwtJ-Z : 222
- set-address Swat command TSwtJ-Z : 222
 - in example TTCL : 330
- setcc Swat command TSwtJ-Z : 223
- set-masks Swat command TSwtJ-Z : 223
- set-repeat Swat command TSwtJ-Z : 224
 - in example TTCL : 330
- set-startup-ec Swat command TSwtJ-Z : 224
- sftwalk Swat command TSwtJ-Z : 224
- Shell commands in makefiles TTools : 474
- showcalls Swat command TSwtJ-Z : 225
- showMethodNames Swat command
 - TSwtJ-Z : 226
- .SILENT TTools : 488, TTools : 490
- skip Swat command TSwtJ-Z : 226
- sleep Swat command TSwtJ-Z : 227
- slist Swat command TSwtJ-Z : 227
- smatch Swat command TSwtJ-Z : 228
- sort Swat command TSwtJ-Z : 228
- Source
 - meaning in makefiles TTools : 472
- Source code
 - displaying TSwatCm : 52
 - displaying with Swat TSwtJ-Z : 229
 - listing with Swat TSwtJ-Z : 227
 - src Tcl structure TTCL : 316
 - stepping through TSwatCm : 52
 - viewing with Swat TSwtJ-Z : 253
- source Tcl command TTCL : 298
- spawn Swat command TSwatCm : 51,
 - TSwtJ-Z : 229
- Special targets in makefiles TTools : 488
- Spreadsheets
 - examining with Swat TSwtJ-Z : 198,
 - TSwtJ-Z : 202
 - printing in Swat TSwtJ-Z : 205
- src Tcl structure TTCL : 316
- srcwin Swat command TSwatCm : 52,
 - TSwtJ-Z : 229
- sstep Swat command TSwatCm : 52,
 - TSwtJ-Z : 230
- Stack
 - changing frames TSwtA-I : 110,
 - TSwtJ-Z : 251
 - examining TSwtA-I : 128
 - examining backtrace TSwtA-I : 88
 - examining data on TSwatCm : 56
 - examining with Swat TSwtA-I : 111
 - handles TTCL : 313
 - thread stack size TTCL : 323
- Standard paths
 - examining with Swat TSwtJ-Z : 231
- stdpaths Swat command TSwtJ-Z : 231
- step Swat command TSwtJ-Z : 232
- step-patient Swat command TSwtJ-Z : 232
- Stepping through code
 - TSwatCm : 52–TSwatCm : 54
 - Asm TSwtA-I : 156
 - Assembly code TSwtJ-Z : 170,
 - TSwtJ-Z : 226, TSwtJ-Z : 232
 - assembly code TSwtJ-Z : 232
 - C code TSwtJ-Z : 230
 - finishing a frame TSwtA-I : 122,
 - TSwtA-I : 123
 - go Swat command TSwtA-I : 138
 - swat events TTCL : 310
- step-until Swat command TSwtJ-Z : 233
- stop Swat command TSwtJ-Z : 233
- stop-catch Swat command TSwtJ-Z : 234
- stop-patient Swat command TSwtJ-Z : 234
- stream Swat command TSwtJ-Z : 235
- string Tcl command TTCL : 299
- Strings in Swat
 - common prefix TSwtA-I : 95
 - computing length TTCL : 292
 - explode Swat command TSwtA-I : 118
- Structures
 - accessing fields in Swat address
 - expressions TSwatCm : 38
 - creating TTCL : 325
 - displaying size in Swat TSwtJ-Z : 205
 - examining with Swat TSwtJ-Z : 258
 - field Swat command TSwtA-I : 120
 - formatting enumerated type values with
 - Swat TSwtJ-Z : 183
 - formatting Swat output TSwtA-I : 82,
 - TSwtA-I : 96, TSwtA-I : 99,
 - TSwtJ-Z : 170
- Suff_FindDeps() routine TTools : 483
- .SUFFIXES TTools : 490



Swat TSwatCm : 33–TTCL : 332
 abbreviations TSwatCm : 71
 address expressions TSwatCm : 37
 address history TSwatCm : 70
 attaching and detaching TSwatCm : 43
 command completion TSwtA-I : 95
 command correction TSwatCm : 69
 command line options TSwatCm : 35
 commands and variables TSwtA-I : 76
 mouse navigation TSwatCm : 69
 online help TSwatCm : 40, TSwtA-I : 84
 SWAT.CFG file TSwatCm : 35
 Tcl (Tool Command Language)
 TTCL : 265
 SWAT.CFG file TSwatCm : 35
 SWAT.RC file TSwatCm : 72
 switch Swat command TSwatCm : 68
 command reference TSwtJ-Z : 239
 .sym files TTools : 437
 Symbol files TTools : 437
 Symbol paths in Swat address expressions
 TSwatCm : 37
 symbol Tcl structure TTCL : 317
 in example TTCL : 328
 symbolCompletion Swat variable
 TSwtJ-Z : 240
 sym-default Swat command TSwatCm : 68
 command reference TSwtJ-Z : 240
 sysfiles Swat command TSwtJ-Z : 241
 systemobj Swat command TSwtJ-Z : 241

T

:T TTools : 492
 table Tcl structure TTCL : 321
 .TARGET TTools : 478
 Target
 accessing hierarchy with Swat
 TSwtJ-Z : 242
 finding with Swat TSwtA-I : 80
 when making TTools : 472
 Target attributes in makefiles TTools : 487
 target Swat command TSwtJ-Z : 242
 targetobj Swat command TSwtJ-Z : 243
 Targets TTools : 472
 .TARGET local variable TTools : 478

tbrk Swat command TSwtJ-Z : 243
 Tcl
 backslash sequence table TTCL : 272
 comments TTCL : 268
 copyright information TTCL : 266
 data structures TTCL : 303–TTCL : 328
 debugging TSwtA-I : 102, TSwtA-I : 103
 defining procedures TSwtA-I : 103,
 TSwtA-I : 104, TTCL : 293
 defining variables TSwtA-I : 105
 loading command files TSwtA-I : 87
 operators illustrated TTCL : 273
 syntax TTCL : 267–TTCL : 277
 table of operators TTCL : 273
 tcl-debug Swat command TSwtJ-Z : 244
 Text
 displaying bytes as TSwtA-I : 91
 displaying text monikers with Swat
 TSwtJ-Z : 210
 displaying text objects with Swat
 TSwtJ-Z : 206
 monitoring line ripple activity with Swat
 TSwtJ-Z : 219
 monitoring undo activity with Swat
 TSwtJ-Z : 249
 text-fixup Swat command TSwtJ-Z : 245
 thaw Swat command TSwtJ-Z : 245
 thread Tcl structure TTCL : 322
 threadname Swat command TSwtJ-Z : 246
 Threads
 breakpoints in not yet created
 TSwatCm : 51
 current within patient TTCL : 315
 displaying name in Swat TSwtJ-Z : 246
 displaying status with Swat
 TSwtJ-Z : 204
 displaying with Swat TSwtJ-Z : 246
 examining registers TSwtA-I : 99
 examining with Swat TSwtJ-Z : 207
 freezing TSwtA-I : 131
 handles TTCL : 313
 monitoring with Swat TSwtJ-Z : 229
 specific breakpoints TTCL : 303
 switching TSwtJ-Z : 239
 switching between TSwatCm : 68
 thawing TSwtJ-Z : 245
 thread Tcl structure TTCL : 322
 waking up with Swat TSwtJ-Z : 257



threadstat Swat command TSwtJ-Z : 246
 timebrk Swat command TSwtJ-Z : 247
 Timers
 displaying with Swat TSwtJ-Z : 249
 handles TTCL : 313
 interrupt TSwtA-I : 153
 printing with Swat TSwtJ-Z : 207
 Timing with Swat TSwtA-I : 100,
 TSwtJ-Z : 247
 timingProcessor Swat variable
 TSwtJ-Z : 248
 tmem Swat command TSwtJ-Z : 248
 tmp files TTools : 437
 Token databases TIconEd : 342
 TOKEN_DA.000 file TIconEd : 342
 top-level Swat command TSwtJ-Z : 248
 Transformation rules in makefiles
 TTools : 481
 Trees
 displaying generic ancestors
 TSwtA-I : 139
 displaying visual ancestors with Swat
 TSwtJ-Z : 255
 displaying visual, with Swat
 TSwtJ-Z : 254
 Displaying window, with Swat
 TSwtJ-Z : 259
 generic TSwatCm : 62
 accessing top of, with Swat
 TSwtJ-Z : 241
 generic, displaying with Swat
 TSwtA-I : 134
 visual TSwatCm : 62
 tundocalls Swat command TSwtJ-Z : 249
 twalk Swat command TSwtJ-Z : 249
 type Tcl structure TTCL : 324
 in example TTCL : 329

U

Ctrl-u Swat navigation TSwatCm : 69
 unalias Swat command TSwtJ-Z : 249
 unassemble Swat command TSwtJ-Z : 250
 unbind-key Swat command TSwtJ-Z : 250
 undebug Swat command TSwtJ-Z : 251
 Unions in Swat

 creating TTCL : 326
 up Swat command TSwtJ-Z : 251
 uplevel Tcl command TTCL : 300
 .USE TTools : 488

V

value Swat command TSwtJ-Z : 251
 in example TTCL : 329
 var Tcl command TTCL : 301
 in example TTCL : 328,
 TTCL : 329–TTCL : 330
 Variables
 assign Swat command TSwtA-I : 85
 Variables in makefiles
 TTools : 476–TTools : 480
 Variables, environment
 accessing from Swat TSwtA-I : 137
 varwin Swat command TSwtJ-Z : 253
 view Swat command TSwtJ-Z : 253
 view-default Swat command TSwtJ-Z : 254
 view-size Swat command TSwtJ-Z : 254
 Virtual memory
 error checking TSwtA-I : 113,
 TSwtA-I : 114
 examining with Swat TSwtJ-Z : 210,
 TSwtJ-Z : 211
 handles TTCL : 313
 handles in Swat address expressions
 TSwatCm : 39
 printing huge arrays TSwtJ-Z : 187
 vistree Swat command TSwatCm : 62,
 TSwtJ-Z : 254
 Visual objects
 displaying ancestors with Swat
 TSwtJ-Z : 255
 displaying instance data TSwtJ-Z : 209
 displaying tree with Swat
 TSwatCm : 63, TSwtJ-Z : 254
 vup Swat command TSwatCm : 63
 command reference TSwtJ-Z : 255

W

wait Swat command TSwtJ-Z : 256



waitForPatient Swat command
 TSwtJ-Z : 256
 wakeup Swat command TSwtJ-Z : 257
 wakeup-thread Swat command
 TSwtJ-Z : 257
 wclear Swat command TSwtJ-Z : 257
 wcreate Swat command TSwtJ-Z : 257
 wdelete Swat command TSwtJ-Z : 257
 whatat Swat command TSwtJ-Z : 257
 Tcl source TTCL : 328
 whatis Swat command TSwtJ-Z : 258
 where Swat command TSwtJ-Z : 258
 why Swat command TSwtCm : 69
 command reference TSwtJ-Z : 258
 Windows
 accessing topmost with Swat
 TSwtJ-Z : 221
 accessing window with grab
 TSwtCm : 62
 clipping region TSwtJ-Z : 196
 displaying tree with Swat
 TSwtJ-Z : 259
 monitoring with Swat TSwtJ-Z : 226
 under mouse TSwtA-I : 152
 wintree Swat command TSwtJ-Z : 259
 winverse Swat command TSwtJ-Z : 259
 wmove Swat command TSwtJ-Z : 260
 WordFlags
 printing flag fields TSwtJ-Z : 195
 words Swat command TSwtCm : 55
 command reference TSwtJ-Z : 260
 wpop Swat command TSwtJ-Z : 260
 wpush Swat command TSwtJ-Z : 260
 wrefresh Swat command TSwtJ-Z : 261
 wtop Swat command TSwtJ-Z : 261

X

:X TTools : 491

Y

Ctrl-y Swat navigation TSwtCm : 69



