MPW 3.0A2 Release Notes

July 14, 1988

Overview

This release note pertains to the following MPW products:

- MPW Development Environment
- MPW Pascal
- MPW C
- MPW Assembler

MPW C++ has its own separate release note. This release note, the MPW 3.0 ERS's, and other tool-specific release notes are the documentation for this release. Differences between the ERS, previous 3.0 development releases, and this release are documented below. If no release note exists for something that has an ERS, assume that the piece has been implemented as documented in the ERS.

MPW 3.0A2 highlights include a symbolic debugger, project management system, and new C compiler. All of the C pieces of MPW, were compiled with this new C. Since MPW 2.0, the C header files have three important changes: every function has a function prototype, the capitalization for routines that pass strings and points has changed, and the definition of Str255 has changed. The capitalization and Str255 changes will require changes to your C source. See the section below titled "Interfaces."

Another change in MPW 3.0A2 may require a change to your Makefiles. The Pascal compiler no longer uses the Load/Dump mechanism. Instead, the compiled object code for each unit is saved in the unit's resource fork. Therefore, dependency rules (in Makefiles) upon Pascal dump files are obsolete. See the section below titled "Pascal."

The A2 C compiler now has function prototype checking turned on. Furthermore, the C compiler is very strict about type compatibility. Be forewarned...

Installation

- **Installation from floppies** - MPW 3.0 now includes an installer disk for installing MPW from a set of floppy disks. Here are the steps for automatically installing MPW on your disk:
    1) drag the "Installation Folder" onto the hard disk where you want MPW installed
    2) launch the MPW Shell found in the "Installation Folder"

(the one you just dragged onto your hard disk)
3) the installation script will run and will prompt you to insert a disk in the internal drive. You can insert the MPW disks in any order and it won't hurt to insert a disk more than once.
4) once the installation is complete, throw away the "Installation Folder," and launch the newly installed shell.

For those not using the installer script, you will notice that the configuration of MPW on the floppies has changed. The files on the disks are in folders that represent their final destination when moved to a hard disk. Pascal, for example, used to be found at the root level of the Pascal disk. Now, however, Pascal will be found in a "Tools" folder on the Pascal disk. This means that there will be duplicate folders across the set of floppies and you won't be able to drag the contents of each floppy onto a hard disk without some conflict.

• **Installation from Spuds/Taters** - simply drag the MPW and SADE folders onto your hard disk.

Reporting Bugs

Please report any bugs you find to the BRC. Please use the latest version of "Outside Bug Reporter," found on the Spuds/Taters server. After completing the bug report, copy the report into the folder "Put New Bugs Here" which is found on:

Zone:	EtherKnott
Server:	BRC Central
Volume:	Bug Jungle

Getting the Latest Stuff

As bugs are reported and fixed in A2, newer pre-beta versions of MPW will be placed on:

Zone:	Development Tools
Server:	AlmostMPW
Volume:	AlmostMPW

Note: this software is for Apple internal use only. Use Backup to find out which pieces are new.

Folder Reorganization

Examples - In order to reduce the propagation of example folders within the MPW folder, all of the example folders have been placed inside a single folder: Examples. This folder contains AExamples, CExamples, CPlusExamples, Examples, LL1Examples, and PExamples.

Interfaces- In order to reduce the propagation of interface folders within the MPW folder, all of the interface folders have been placed inside a single folder: Interfaces. This folder contains the folders: AIncludes, CIncludes, PInterfaces, RIncludes, and AStructMacs. The various interfaces Shell variables are, of course, set up correctly by the Startup script. If you have any hardcoded paths in scripts, however, they will have to be changed.

Libraries- In order to reduce the propagation of library folders within the MPW folder, all of the library folders have been placed inside a single folder: Libraries. This folder contains the folders: CLibraries, Libraries, and PLibraries. The various libraries Shell variables are, of course, set up correctly by the Startup script. If you have any hard-coded paths in scripts, however, they will have to be changed.

Debuggers & ResEdit- Since MacsBug, SADE, and ResEdit are now separate products from MPW, their folders are no longer found in the MPW folder. In fact, the Debuggers and Applications folders no longer exist. The SADE, MacsBug, and ResEdit folders can be placed anywhere on your hard disk.

Release Notes

⁄IPW Shell

The Shell enhancements are documented fully in the Maintosh Programmer's Workshop 3.0 Shell ERS.

3.0 A2 Enhancements Included:

The Shell variable {MPW} is now {ShellDirectory} rather than {Boot}MPW:. This makes it easier to run MPW from a non-boot volume.
Tools are now opened read-only. They may be run in a shared environment (e.g. from a file server).
The Startup script will now execute any file in the shell directory named "UserStartup•≈" (in addition to UserStartup).
The **Files** command has an extra option -h. This option is for not printing the directory headers when multiple directories are listed. This allows the output of files to be used in pipe without worrying about a directory listed twice (with the -r option) or worrying about blank lines or the directory name.
The **MoveWindow** command has a new option -i. This option is for ignoring errors on the position of the window. This allows windows to be moved completely off screen. Once a windows is moved off the screen, it can only be moved, closed, or zoomed through commands.

Pre 3.0 A2 Enhancements Included:

Directory Path
Numeric Variables

Background operation of Tools
Request -q
Close -c
FAccess - Selection information
Read only check box in open dialog
Current window size and position information (SizeWindow, MoveWindow)
RotateWindows Command
Format Command
New shell variables: {Font}, {FontSize}, {Tab}, {AutoIndent}, {SearchWrap}, {SearchBackward}, {SearchType}
The tab limit has been increased to 80.
The line length limit has been increased to 256.
Horizontal scrolling goes faster.
Option-Enter will invoke Commando.
TileWindows and StackWindows have options for windows and area for tile/stack.
The Date command has -n and -c options for date arithmetic.
Opening zoomed windows will zoom to current screen.
Locked and read-only will open without the dialog box. An icon in the bottom of the window will display the locked or read-only status.
Selecting text by matching delimiters has changed: You may abort by pressing cmd-., and if a matching delimiter is not found, the current delimiter will be selected.
ZoomWindow without any options will toggle the window size just like clicking in the zoom box.
Σ (Option-W) will direct both stdout and stderr to the same place.
Evaluate supports decimal, hexadecimal, octal, and binary radices.
New windows will be opened according to rectangle {NewWindowSize}.
Zooming will be done according to rectangle {ZoomWindowSize}.
The TileWindows and StackWindows menu commands may be customized with {TileOptions} and {StackOptions} variables.
New Position command.
New Flush command. This command will clear any tools that are in the cache.

See the Shell ERS for more specific information.

Projector

Read the Projector ERS, release note, and manual pages for complete details on how to use the new project management system. Projector, by the way, is actually part of the MPW Shell.

Note: if you used projector in 3.0A1, you will need to convert your project. Be sure to read the "Converting Your Projects" document.

Pascal

• The 3.0 Pascal compiler no longer supports the '{$LOAD}' facility. Instead, the compiler automatically builds a 'pre-compiled' version of the symbol table for each unit and puts this into the resource fork of the file containing the unit. On subsequent compilations, the compiler will load this resource instead of compiling the unit. The compiler will not use the resource if the modification date of the file is later than the date stored when the resource was created, or if compile time options that were in effect when the resource was created have changed in such a manner as to invalidate the resource. The '{$LOAD}' syntax is still supported, but ignored – if compiler progress information is requested, the compiler will state that the use of the feature is 'obsolete but harmless'.

Some users will have included dependencies for '{$LOAD}' files in their makefiles. These can now be removed, but they are harmless since they simply restate what the compiler does automatically.

There are three new command-line options that help support the new feature :

-NoLoad	don't use or create any symbol table resources
-Clean	erase all symbol table resources
-Rebuild	rebuild all symbol table resources

• The following items have not yet been implemented or are under construction:
  - Global Data > 32k is not supported.
  - Forward references for records and objects must still be resolved within a single TYPE block.
  - The 'nolines', 'novars' and 'notypes' options for SADE do not work.

• Chars can be used where strings used to be required (e.g. someString := CHR($0D);)

C Compiler

• Assignment compatibility for pointers is strictly enforced.

• Type checking for function prototype arguments is <u>strictly</u> enforced. For example:

```
extern pascal short GetCtlValue(ControlHandle theControl);
Handle h;             /* not a ControlHandle */
short  val;

val = GetCtlValue(h); /* will generate a compiler error */
val = GetCtlValue((ControlHandle)h);        /* ok */
```

• Some of the compiler options are different than the Greenhills compiler. See the "C Manual Page" (in the Release Notes folder) for a complete list of the C compiler options. The C compiler's Commando dialog now shows all of the compiler's available options.

• The `volatile` keyword has been implemented as meaning "not `register`."

• For 3.0, the C compiler's calling conventions are the same as the Greenhills compiler with four exceptions:

  1) the underlying mechanism for functions that return structures or unions is incompatible between the two compilers (i.e.

mixing object code in this case will break).
2) unlike the Greenhills compiler, no global variables are ever generated by functions returning structures or by switch statements.
3) functions that return results that are smaller than a longword return the smallest possible result in D0 (instead of extending the result to a long).
4) the C compiler now considers D2 to be a scratch register.

- If at all possible, recompile all C source with this compiler to avoid mixing pre-3.0 object code with 3.0 object code.
- If your C source is using pre-3.0 C header files, be sure to convert your C source with the CCvt script. There is more information about the changed interfaces below.

- Here is a list of known bugs or features under construction:
  - The 'U' and 'F' constant type modifiers are not completely implemented.
  - The 'nolines', 'novars' and 'notypes' options for SADE do not work.
  - SADE output is limited by static internal buffer size.

Interfaces

The major change to the interfaces for MPW 3.0 was to change capitalization conventions in CIncludes for those functions which use Points or strings. This was a major change which was specified in the MPW 3.0 ERS. Functions calling glue to convert C strings to Pascal strings or dereference Points are now spelled with all lower case. The inline versions of those function calls, which do no conversion, are now spelled with mixed case spelling to match "Inside Mac". In order to easily convert source code to these new conventions, we have written a script, CCvt (in MPW:Scripts), which calls the Canon tool to change the sources. CCvt first duplicates the original source as a backup procedure. It then uses two canon dictionaries located in the Tools folder, CCvtUMx.dict and CCvtMxL.dict, to first change mixed case spellings to all lower case and then to change all upper case spellings to mixed case conventions. CInterface.o and other libraries changed so the linker would find the glue necessary for these lower case C functions.

See the Interfaces release note and the Pre-A2 Interfaces note for much more information.

Linker Tools

See the Linker Tools release notes.

Libraries

Also see the Libraries release notes.

• In the C library's printf formatting string, the meaning of %p has changed. Before this release, it meant read/print a pascal-style string. In order to conform to the ANSI spec, however, %p is now used to print the value of a pointer. %P is used to read/print pascal-style strings.

• The library Stubs.o is new. MPW tools can save about 4300 bytes each by linking with this library BEFORE linking with Runtime.o or CRuntime.o; standalone applications should not link with this library.

Pascal bugs fixed:
• Error message for set range checking is no longer garbled.
• IOResult no longer reports error -1025 (an AppleTalk error number) when string overflow is detected, and when READ expects an integer, but gets no numeric value. IOResult now reports error 34 in these circumstances.

Pascal changes:
• Added support for error manager in startup sequence.

C and Integrated Environment changes:
• Added support for error manager in startup sequence.
• F_GSELINFO and F_SSELINFO now supported in all languages.
• mktemp() is now implemented. To support mktemp(), we also have getpid() and access(). These functions will all be documented once it is clear how they fit into the ANSI environment.

Choose Tool

Choose is a new MPW tool that allows you to mount servers and select LaserWriters from the MPW environment. See the ERS for more information. Choose now works with AppleShare version 1.1 and 2.x. Choose no longer complains about "no bridge found" if you happen to be in a zone-less network.

Commando

Commando has a simple built-in editor that allows controls to be moved around and resized and labels and help messages to be edited. See the Commando release note.

Make

Changes have been made to the way variables (macros) are treated:

• Exported shell variables are automatically expanded by Make so that variables such as {AIncludes}, {MPW}, etc. can be used to express file names. Recently, however, {Status} was made an exported variable, which had the unfortunate side effect that if {status} was referenced in Make build rules it would get emitted as "0" (the value of status at the time of running Make). The 3.0 version of Make will still automatically define exported variables for use in dependency lines, but will now not expand exported variable references when they appear in build rules.

- Variable references in variable definitions were previously required to refer to already-defined variables, since the variable references were expanded at definition time. Variable references in variable definitions are now not expanded until the time of use, which allows some new behaviors which were not permitted before. (All old makefiles will continue to behave as they used to.) Now variable definitions can refer to dynamically defined variables such as {targ} which don't take on values until build rule expansion time, allowing some clever effects not possible previously.

None of these changes should change the behavior of any currently working make files, although how errors are reported for bad variable definitions has of necessity changed.

Print Tool

Although there is no ERS for the print tool, there is a new print option:

    -ps filename

This option allows one to include a file of postscript commands to be sent to the laserwriter prior to printing the file. See print's commando help message for more information.

Rez
DeRez
Types.r
SysTypes.r

- All features have been implemented as documented in the Rez Tools ERS (you will need WriteNow to open the ERS).
- In Types.r, the following type declarations have changed:

'cicn', 'ppat', 'crsr'	- redefined to use labels (not changed since D1)
'SIZE	- now supports MultiFinder and Switcher
	- two new flags have been added

- In SysTypes.r, the following type declarations have changed:

'scrn'	- flags were in the wrong order
'FONT', 'FOND'	-redefined to use labels
'snd '	- newly added
	- added synth values
'vers'	- newly added
KSWP'	- added more specific bit information

- A new resource template, Pict.r, allows you to DeRez PICT's (both types 1 & 2)

LL1Generator

MPW 3.0 includes a parser generator tool (similar to YACC) called LL1Generator. See the ERS and the LL1Examples folder inside the Examples folder.

DumpFile

See the DumpFile ERS.

Changes in this release from previous releases:
- The -c option has been renamed -w (for width).
- A -g nn option has been added to permit groupings other than the default.

CreateMake

- Several changes have been made to CreateMake and to its Commando window. To the existing options for program type: Application, Tool, and DA, has been added a fourth option: CR, which is an abbreviation for Stand-Alone Code Resource. If CR is specified, the parameters -m mainEntryPoint and -rt resourceType are mandatory. The commands -c creator and -t type (meaning file type) are optional for CR and -c creator is optional for Application.
- Further changes are that the makefiles produced by CreateMake now contain lines of the form:

```
SOURCES = <all source files>
OBJECTS = <all object files>
```

- The diagnostics for calls of CreateMake with improper parameters have been improved.
- CreateMake does not yet support a 68881 code generation option.

SetVersion

SetVersion now supports the 'vers' resource as documented in TechNote #189. See the new manual page in the release notes.

Projector Alpha 2 Release Notes

Authors: Jeff Parrish
Bob Etheredge
Peter Potrebic
John Dance

Date: July 6,1988

What is Projector, and how the heck do I use it

Projector is a collection of built–in MPW commands and windows that help programmers (both individuals and teams) control and account for changes to *all* the files (documentation, source, applications, etc.) associated with a software project.

Which ERS To Use

The exact definition of the built–in MPW commands and windows is specified in the July 6, 1988 Projector ERS. This ERS is a complete, accurate description of Projector, it is not just a list of the differences since the Alpha 1 ERS — so throw away any old copies you might have lying around. In addition, the ERS describes the Projector model for controlling project files, and details many of the issues related to that model.

The Projector team *strongly* suggests that before you use Projector, you at at least read the ERS overview and section dealing with the components of a Project.

The Projector team, as well as other members of the Development Systems Group, have been using Projector for several months. Projector has been used both as a network based project control system shared between several people, and as a local HD based system used by single individuals.

Performance

Performance will continue to be one of our highest priorities between alpha 2 and beta (assuming no one finds the bug we left in). Significant performance enhancements have already been added between the alpha 1 and alpha 2 releases. Performance has remained essentially constant between alpha 1 and alpha 2 in spite of the fact that the data compaction that was added for alpha 2 requires a considerable amount of processing and I/O.

The sections below represent the accumulated wisdom of Projector's initial users; what they found to be particularly useful, and what they could easily do without. Since the Projector team would definitely like to improve Projector (and this list), please let us know what you think we can improve, as well as what we are doing right. If there is something you don't like about Projector (I know it's hard to imagine), please help us out by taking a minute to think of how it could be improved.

Comments and questions can be directed to: Jeff Parrish x2395, Bob Etheredge x6250, or Peter Potrebic x6494.

Important Update Info

What's new since Alpha 1

The Alpha 2 Projector is incompatible with Alpha 1 projects. The "Convert" folder (which can be found on the AlmostMPW server in the Development Tools zone, in the AlmostMPW volume) contains the scripts and tools needed to make your existing projects compatible with the new Projector (and save loads of disk space). The details of how to convert your projects are in the release note titled "Converting your projects".

Projector now saves 'TEXT' files in compressed form.

A "Touch Mod Date" check box was added to the "Check In" and "Check Out" windows. The default for the "Check In" window is to leave the revision's modification date untouched. The default for the "Check Out" window is to touch the revision's modification date.

The "-touch" and "-noTouch" options were also added to the "CheckIn" and "CheckOut" commands respectively.

Modifying Read–Only Files

The new Projector command "ModifyReadOnly", was added to allow users to modify files that were checked out as read–only. The primary use of this command is to allow users who don't have access to a project (e.g. working at home), to edit the read–only files that were previously checked out. Once this command has been run on a file, the user may treat the file as though it were checked out for modification—with one exception: you will not be able to check a modified read–only file into a project if some-one else happened to have created a revision more recent than the revision you modified. The syntax is:

```
ModifyReadOnly file.
```

The Shell now displays all the appropriate Projector icons in its editing windows. The Shell used to display only the "read–only" and "locked" icons for the files that belonged to Projector.

A "-cancel" option was added to the CheckIn command to correspond to the "Cancel Checkout" button in the "Check In" window.

Projector will automatically continue to try to open a project if the project is currently being accessed by someone else. A "command–." will discon-tinue the automatic retry.

Several changes have been made to the Projector windows to help differ-entiate their various states. For example, the window titles change to indi-cate when the window is displaying information.

A logging mechanism has been added to Projector to keep track of all the commands which delete information from a project. The "-log" option to the ProjectInfo command can be used to list a project's log.

The new options, "-newer" and "-update", have been added to the CheckOut command. The "-newer" option will check out all read–only files that you either don't currently have or are newer than your existing files. The "-update" option will check out only the read–only files that are newer than your existing files.

A "-m" option has been added to the ProjectInfo command and can be used to list information about all the revisions that are checked out for modification.

'Jseful Scripts

Two new scripts/commands have been added to make manipulating 'ckid' resources a little bit easier: "OrphanFile" and "TransferCkid". It is important to keep in mind that you will only need these scripts under exceptional circumstances. For example, if you duplicate a file that has been checked out from a project, you will end up with two files that both think they have been checked out. OrphanFile can be used to clean up the duplicate file.

An example of when you would use TransferCkid, would be if you were using Projector to maintain the history of a file where each revision was generated from scratch (e.g. most object files, Microsoft Word documents, etc.). Since the new revision would not have a 'ckid' resource, you would have to check out the latest revision of the file for modification, use TransferCkid to move the 'ckid' resource from the checked out revision to the generated file (to make Projector think that the generated file was checked out for modification), and then delete the revision you checked out. What you are then left with, is a new revision of the file that can be checked back into the project.

Two other utility scripts, CheckOutActive and CheckInActive, have been supplied in an example project in the MPW Examples folder. To access these scripts, execute the following commands:

```
MountProject "{MPW}Examples:Projector"
CheckOutDir Projector∫Utilities "{MPW}Scripts:"
CheckOutDir Projector∫Commands "{MPW}Scripts:"
```

and then use the "Check Out" window to look at the scripts in the example projects and decide which ones to check out. Both CheckOutActive and CheckInActive are intended to be used as user–defined menu items. An example of how you might set up your menus would be to execute the following commands:

```
AddMenu Project '(-' ''
AddMenu Project 'Check Out Active' 'CheckOutActive'
AddMenu Project 'Check In Active' 'CheckInActive'
```

Special features that make life wonderful.

Holding the option key down while you choose the "Check Out" button in the CheckOut window will open any TEXT files after they are checked out.

Be sure to add a comment (or task) when you check a file out from the CheckOut window. It gives other people the opportunity to find out what changes you are currently making to the file.

Canceling Checkouts

If you check a file out by mistake, you can easily cancel the check out by selecting the "Cancel Checkout" (of course) button in the CheckIn window.

If you check out a file for modification and then lose it (delete it, send it to your aunt Millie, etc.), you can cancel/delete the checkout by using the -checkout option to the DeleteRevisions command. Think three or four times about this before you actually do it though. If you cancel/delete the checkout and then find the file, Projector will disavow any knowledge of the file or any of its IM force.

If you find yourself working on a shared file, but doing something a little (or a lot) off the beaten path, create a branch! You can create a branch either when you check a file out, or when you check a file in. To create a branch when you check the file out, simply check the "branch" check box before you check the file out. To create a branch when you check in the file, select the "Revision..." button in the CheckIn window and then check the "Create a Branch" checkbox. Branches allow you to work on a tangent to the main development effort without affecting anybody else. For example, if you wanted to add and test out a new source file, you might check out the group's makefile on a branch in order to add the dependencies on the new file.

CheckOutDir Short Cuts

The CheckOutDir command gives you the ability to have Projector automatically direct files checked out from a project to any directory you choose. If your directory structure (and names) match your project structure (and names), you can use the following handy CheckOutDir command to automatically map all your projects to their corresponding directories:

```
CheckOutdir -r -project TheRootProject∫ TheRootDir
```

The "Select Newer" button in the CheckOut window is a great way of ensuring that you have the latest copies of all the files in a project.

Changing the current project (whether by using the Check In or Check Out windows, or by using the Project command) will cause the check out directory to be displayed in the Check In window. In the case of the Check In window, this means that changing the current project will automatically adjust the list of files displayed according to the CheckOutDir you have set for that project.

Adding To A Symbolic Name

To add or delete a specific revision from an existing symbolic name, you can get the current definition of the name (the -s option is helpful in that each component of the name is listed on a separate line), and then edit the specific entries you're interested in. Note: symbolic names are tied to projects, *not* revisions. For example, the following three commands:

```
NameRevisions george file.c,3 file.h,5
NameRevisions george file.c,1
NameRevisions -s george
```

will print out

```
NameRevisions george -project MyProject -user Me ∂
file.c,1
```

not

```
NameRevisions george -project MyProject -user Me ∂
file.c,1 ∂
file.h,5
```

To delete symbolic names, use the DeleteNames command.

Special features that make life not so wonderful.

Think twice before you use the DeleteNames command with the -public option. Once you have deleted a public name, it is gone for good—like totally. If this happens to you, you can use the -log option to the ProjectInfo command to find out what the name was, and then manually recreate it.

The NameRevisions command with the -public option is also dangerous (for the same reason as above) because it will replace any previous definition for the same name.

There is no automatic merge for revisions. To merge two revisions, you can check the revisions out to two different directories and then compare and merge them the same as you would any two files today.

If you delete a project without deleting all the files checked out from it, the files will continue to think that they belong to the project long after the project is gone. A symptom of this is having files in the "Check In" window listed with an icon showing a document with a question mark (this will also occur if the project isn't mounted). Files in this state cannot be checked into any projects. A file with such an icon does not necessarily mean that its associated project has been deleted. To find out if it has, click the big question mark button to switch the window to an information view, and then select the file in question. If the project listed for that file does not exist, your file has been orphaned. If the project listed for that file currently exists, make sure that the "current project" in the Check In window is the project listed for the file. If the file still shows the same icon, you probably deleted the original project and then created a new one with the same name.

If you find out that you either deleted a project or deleted a project and then created a new one with the same name, you will need to disassociate the file from the original project by using the "OrphanFile" script. **Caution: make sure that you only orphan the files associated with a deleted project.**

Converting your projects

The following describes the process of converting projects created with the old Projector (MPW 3.0 Alpha1) into projects compatible with the new Projector (MPW 3.0 Alpha2). This conversion is necessary because the internal format of the project changed to support deltas and the logging mechanism.

Please read this entire document before proceeding with the conversion.

Steps to take:

1. Please read this entire document before proceeding with the conversion.
2. Make a backup of your projects.
3. All modifiable files must be checked in and all read-only copies must be deleted. Before conversion takes place the tool makes sure that no files are checked out for modification and conversion will terminate if any files are checked out.

   Execute the following command (in the Alpha1 Shell) on each project tree:

   ```
   projectInfo -project Project -m -r -revisions
   ```

   This command will list all revisions that are checked out for modification. The files must be checked in before continuing.

   Deleting all checked out files is necessary because the format of the ckid resource changed, there by invalidating files checked out using the Alpha1 Projector. The script “FindOldFiles” (in the “Convert “folder) will recursively search directories and list Delete commands for all files that have a ckid resource.

4. Make a copy of the “Convert” folder on a local hard disk.
   This disk will be the destination disk - the new projects will be created within the “Convert” folder. You will need enough space on this disk to accommodate the new projects. Previous conversion have reduced the size of projects from 2 to 5 fold depending on several parameters. Remember that only TEXT files are compressed – projects with non-TEXT files will not shrink as much as projects with only TEXT files.

5. Manually save all public names
   This needs to be done because the conversion tool ignores names.

   ```
   NameRevisions -project Project -r -public >> savedNamesFile
   ```

   Execute the above command on every project tree. This will save the definition of all public names. The names will be restored after the new projects are created.

6. Set the current directory to “Convert” on the destination disk.

7. The {User} variable must be set and exported, preferably to someone's name.

8. Convert old projects by running the following command:

   ```
   projectCvt Projects...
   ```

   The list of projects should contain the full HFS pathname of all your old root projects. The conversion process will convert each of the project trees and when it finishes the new project trees will be inside the "Convert" folder.

   The first part of the conversion process is the validation stage where all the projects are inspected to make sure that no file is checked out for modification. Depending on the number and size of the projects this stage could take a few minutes. Please wait until this stage is over (the conversion tool will print out an appropriate message) before leaving the machine because conversion will terminate if any files are checked out.

9. After the conversion has successfully completed replace the old projects with the new projects – Copy the projects out of the "Convert" folder to replace the old projects.

10. Mount all the new projects and execute the file *savedNamesFile* in order to restore the public names.

Caveats and Notes:

1. Think about what Shell you a running. In order to checkin files into your existing projects you must run the old Shell (pre Alpha2). The conversion process needs to be run in the new Shell.
2. The conversion process adjusts your Mac's clock so don't be alarmed if you noticed the time changing wildly. The script will restore the proper time when completed. Unfortunately, if the script terminates because of some error (e.g. out of disk space) the proper time will not be restored.
3. Be patient - the conversion process is slow. As a rough guideline it took 7.5 hours to convert a 25 megabyte project tree on a local disk.
4. We recommend that the conversion place place locally rather than over the network. Check to make sure that all files are checked in before copying the project off the server. In this way people can easily check in the appropriate files. Then make sure to change the access privileges so that people can't inadvertently modify the old project because these changes won't be reflexed in the new project.

Macintosh Programmer's Workshop
Project Management System ERS
"Projector"

Bob Etheredge, Jeff Parrish, Peter Potrebic
July 6, 1988

Changes (4/14/88):

- Added the modifyReadOnly command
- Updated the window pictures and descriptions
- Described the logging mechanism

Changes (4/14/88):

- ObsoleteRevisions has been removed from the "will be done for MPW 3.0".
- Added Task and Comment fields to the 'ckid' resource and changed it to lower case. The user can now add Task and Comment information to a file without requiring the file's project to be currently mounted.
- Removed the ProjectInfo window as a separate entity, by integrating its functionality into both the CheckIn and CheckOut windows. To get project information, just press the Question Mark button in the CheckIn or CheckOut windows.
- Replaced the {Project} variable with the Project command.
- The project option, used in most of the command examples has been renamed from -p to -project.
- References to the "Log" file have been removed.
- The new option in the CheckIn command has been renamed from -n to -new.

Changes (1/25/88):

- The resource forks of all files are saved on a per revision basis. Previously, the resource fork of text files was not saved.
- Updated the screen shots of the three projector windows (The "Project Info" window has not been updated). The pictures are of the windows as they exist in MPW 3.0 D3. In some places the descriptions of the windows refer to not yet implemented features.
- MergeRevisions has been removed from the "will be done for MPW 3.0" to the wish list appendix.

Table of Contents

Introduction

Purpose of this Document

This document describes the Macintosh Programmer's Workshop (MPW) Project Management System named "Projector".

Product Definition

Projector is a collection of built-in MPW commands and windows that help programmers (both individuals and teams) control and account for changes to *all* the files (documentation, source, applications, etc.) associated with a software project. Projector can be used to coordinate changes among a team of programmers, and to maintain a history of project revisions.

Hardware Compatibility

Projector will run on the same machines that are supported by MPW 3.0, i.e. the Macintosh II, the Macintosh SE and the Macintosh Plus.

Software Compatibility

Projector requires the presence of the Macintosh Programmer's Workshop 3.0. It will not run outside MPW either as a Macintosh application or desk accessory.

Syntax Notation

The Projector ERS uses the same notation as used by the MPW 2.0 Reference manual (the description can be found on page viii of the Preface). A glossary term appears in boldface only the first time it is mentioned.

There are two types of special paragraphs in this document:

☞ Paragraphs marked with a pointing finger explain design decisions or contain a more in depth explanation for a point discussed in the text.

🛑 Paragraphs marked with a stop sign raises an issue that has not been resolved. The Authors would appreciate extra thought and comments on these areas.

Overview

During the evolution of a project each team member invariably makes numerous changes to the source and documentation files that comprise the project. At present, MPW has no integrated facilities to help teams manage the files comprising a programming project. Projector is designed to substantially ease this task by providing an easy to use yet powerful facility for file management that is useful to both the individual programmer working on a small project and a team of programmers working on a large and complex set of programming projects.

Projector organizes the programmer's files into **projects** which can either be stored locally on a hard disk or floppy or remotely anywhere on the AppleTalk network. Each project contains zero or more files. In addition, projects may contain other projects. This last fact is of key importance, since it allows large projects to be broken down into subunits yet accessed as a whole by those outside of the immediate programming team, e.g. testing, software configuration management, etc.

When the programmer wishes to work on one or more files, he or she selects the appropriate project and "checks out" the files needed in the same fashion that books are checked out from the public library, although Projector distributes both read-only and modifiable copies of the "books". This creates a copy of the file(s) that the programmer can modify. Projector remembers the fact that the file is checked out and denies access to anyone attempting to modify the checked out files. Of course, Projector has a mechanism where more than one person can modify the same file simultaneously; this will be fully discussed in a later section.

The programmer can check in the file at any time, although files are normally checked in once modifications are complete and tested. This new file is now available to anyone on the team.

Each new copy of a file is referred to as a **revision**. All revisions to a particular file define its history. Besides supporting a single sequence of revisions to each file, Projector also allows alternative revisions to be created. This feature is called revision **branching**. Branching allows:

- old revisions to be modified.
- several programmers to work on the same revision of a file simultaneously.
- parallel lines of development. The alternate lines of development may be experimental in nature.

As programmers iterate through the checkout – checkin process, they are encouraged to document all the changes and the reasons for the particular changes. This allows the project's current status and history to be easily retrieved by all team members.

Features

- Projects can be organized into a hierarchy of projects.

- All revisions to a file are saved. Each revision is uniquely identified by its file name and **revision number**.

- Allows non-text as well as text files to be stored in the project: e.g. Word, Paint, and executable files.

- Revisions made to text files are stored in a compact format.

- Access by multiple **users** is supported. AppleShare can also be used to assign and control access privileges.

- Flexible naming allows revisions to files to be identified by symbolic name as well as by file name and revision number.

- The entire history and status of all of the files in the project can be conveniently and accurately displayed. **Comments** can be saved with revisions, files, and projects. Projector also associates a **task** with every revision of a file.

- A command line interface is supported. This allows project requests to be embedded in MPW shell command files.

- A window-based interface is supported. This allows for convenient and easy browsing and access to projects.

Limitations

- All files in a project must have unique names.

- Revisions to non-text files are not compressed.

- Commas are not allowed in file names.

Using Projector

Components of a Project

Projects

A project consists of a **project name**, an **author**, some text describing the project, a **project log**, a set of files belonging to the project and zero or more subprojects which also are projects in their own right. The author is the person who created the project. Projects can reside locally on a user's disk or can be placed on an AppleTalk file server to facilitate access by more than one user. AppleShare can be used to assign privileges to different users.

Projector has a Project command that determines the **current project,** i.e. the project the user is currently working on. Projector assumes all Projector commands pertain to the current project unless told otherwise.

The project log keeps a record of all actions that delete information from the project, including deletion of revisions and creation and deletion of public symbolic names. The record that is kept includes the name of the person who carried out the action, the date and time, and exactly what was done.

The **project directory** of a project is the directory where the project resides, and it is created when the project is created. This directory is the same for all users of the project. All the revisions to all the files and all other Projector information is kept in the project directory within the **project file** (called ProjectorDB). Nested projects are also kept in this directory as subdirectories. Every user has a **checkout directory** for each project, this is the directory where, by default, Projector will place checked out files. The checkout directory can be changed with the "CheckOutDir" command.

Each user can select one or more projects to access by using the "MountProject" command. Selecting a project makes it and all its nested projects accessible to the user (see the following discussion on nested projects). Projects can also be removed from the list with the "MountProject" command. Typically, the UserStartup file contains a series of "MountProject" commands that connects the users to a set of projects. Simply mounting a volume does not give a user access to the projects that are contained on that volume. This would be undesirable since many projects may not be of interest to the user. The "MountProject" and "CheckOutDir" commands allow users to customize their own project name space. The location of the project directory is the same for all users, but the checkout directory can be different for each user. For example, Bob and Peter both access the sort project, but they have different checkout directories (see figure 1). When Peter checks out files they go, by default, to Rambo:work:sort, where as Bob's files go to "hd:MPW:Tool Projects:sort tool:".

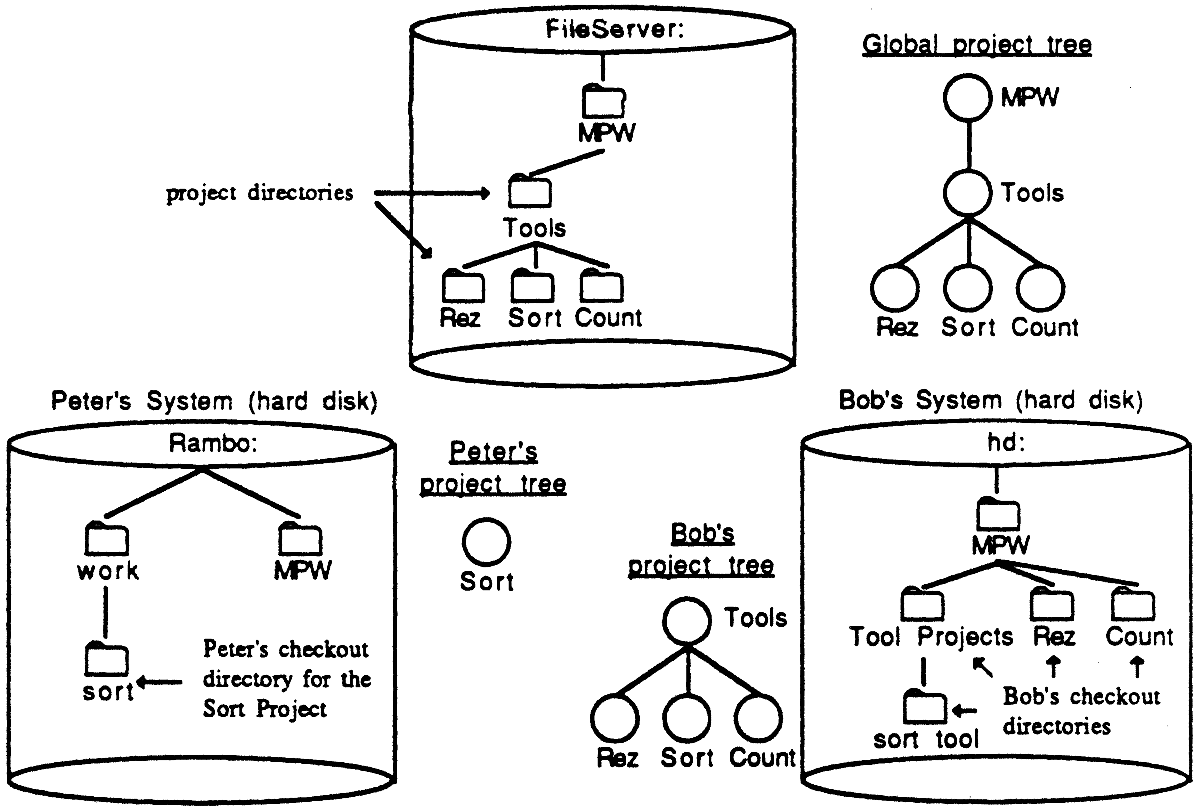


Figure 1 - Example configuration.

Peter's UserStartup could contain the following commands:

```
MountProject FileServer:MPW:Tools:Sort
CheckOutDir -project Sort∫ Rambo:work:sort
```

Bob's could contain:

```
MountProject FileServer:MPW:Tools:
CheckOutDir -project Tools∫ "hd:MPW:Tools Projects"
CheckOutDir -project Tools∫Sort "hd:MPW:Tools Projects:sort tool"
CheckOutDir -project Tools∫Rez hd:MPW:Rez
CheckOutDir -project Tools∫Count hd:MPW:Count
```

(See the MountProject and CheckOutDir manual pages for more information and examples.)

Most Projector commands require a project name as a parameter. The command line interface to Projector supports two ways to specify the project the command will affect. The order of precedence (from greatest to least) is:

1. Use the project specified on the command line.
2. Use the current project as specified by the Project command. By setting the current project to the name of a particular project the user does not need to specify that project with every command.

If Projector cannot determine the project to access, an error is reported and the command is aborted. When using Projector interactively the project is selected by selecting the desired project in a manner that is similar to opening a folder in Standard File.

Projector supports nested projects. A series of related projects, such as all the projects within the MPW product can be configured as a hierarchy of projects. Team members can then access the project structure on the level they choose, very similar to the way people use HFS. See figure 2 for a sample project hierarchy.

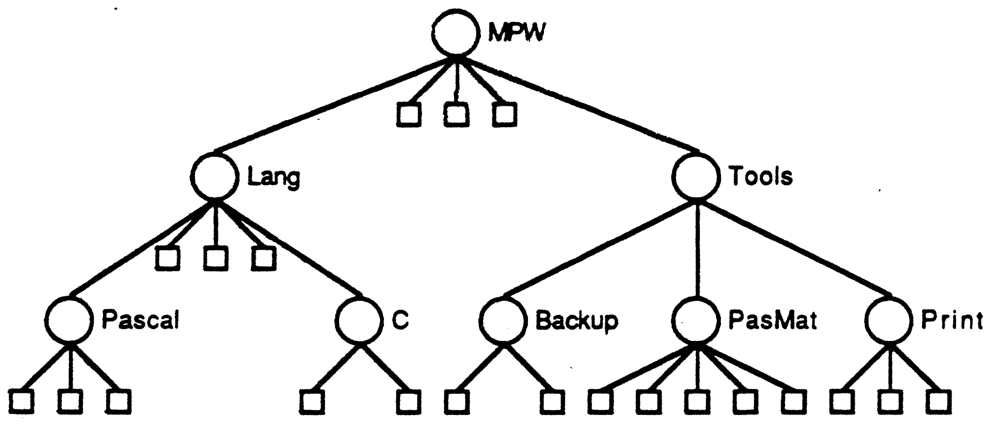


Figure 2 - Sample project hierarchy.

In Figure 2 the MPW project is a highest level project, since it does not have a parent project. Projects are drawn as circles and files are smaller boxes. Just as a person can mount several volumes, there can be several mounted projects. However, Projector does not allow mounted projects to have identical names. The MountProject command is used to add projects to the root project list (called "mounting" projects). In the figure above mounting the MPW project gives the user access to all the projects in the tree.

Projects are named in a similar fashion to directories, only the integral character ('∫', option-b) is used as the name separator. However, Projector requires full path names at all times. Partial project path names will not be supported. Similar to HFS, an integral character ('∫') at the end of a project path is optional.

☞ Projector does not use colons as project pathname separators to avoid confusion with HFS pathnames. Some commands, NewProject for instance, accept both HFS and project paths as parameters. Since the separator is different there will not be any confusion as to what the parameter represents.

☞ Integral characters are not allowed in Project names for the same reason that colons are not allowed in HFS paths.

Files

Each file in a project consists of a name, an author, text describing the file, a record describing the current state of the file, i.e. who has checked out the file, etc., and all of its revisions. The author of a file has no special privileges; this field is basically used for accounting purposes - tracking and assigning portions of a project to different team members.

Projector can be used on all types of files, i.e. TEXT, APPL, Word documents, etc. The only difference between text files and non-text files is that revisions to non-text files are saved, but are not compressed.

Apart from this difference there is no distinction between text and non-text files. Users can check out read-only copies of non-text files, or check out such a file for modification and then check in a new revision. Revisions of non-text files can also be named and deleted.

File names within a project must be unique. Also, commas are not allowed in file names because commas are used to separate file names and revision numbers.

Revisions

Each time a programmer checks in an updated copy of a file a new revision is created. As changes are made and the number of revisions grows a **revision tree** forms that traces the history of the file. By accessing various portions of this tree a user can retrieve, inspect, and compare any of the previous revisions of a file. Projector also allows old revisions to be deleted when the revisions are no longer of interest.

Once a revision is checked out for modification it is **locked** preventing a second modifiable copy of that revision from being checked out. However, checking out a modifiable copy of another revision is okie dokie, and so is checking out a read-only copy of the locked revision. If a user needs to check out a revision that is already locked Projector can create a new branch for this new copy. The user can then manually merge the changes to synchronize the file.

Each revision of a file has a revision number, a creation date (i.e. when it was checked in to the project), a comment describing the reason for the revision, a task, an author of the revision, and a compacted copy of the file itself. The task is simply another place to record information about the revision. The comment field is intended to document the specific changes to a file while the task field could be used to tie different revisions, perhaps across several files, together. For example, implementing a feature might require several changes to each of three files. Each revision might have a different comment, but the tasks for all the revision could say "enhancement X". The task makes it easier to look at the history of a project and determine what changes were made to accomplish various tasks. Projector has reserved the shell variable "Task" as a place to maintain the current task. When using the "Check In" and "Check Out" windows **{Task}** is placed in the "Task:" field by default. There is no default task when using the command line interface (see the CheckIn and CheckOut command pages to see how a task can be specified).

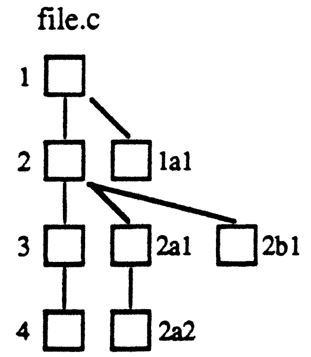


Figure 3 - Revision tree.

Revisions are normally numbered in order, i.e. 1, 2, 3, ..., 99, 100, 101, etc. However, the user can use major/minor numbering instead, i.e. 1.1, 1.2, 1.3, ..., 1.99, 1.100 , 1.101, ...2.1, 2.2, etc. When a new revision of a file is checked in Projector will automatically increase its revision number by one, i.e. 4 to 5, or 4.9.2 to 4.9.3. The user can override this action by specifying a different revision number. The only restriction is that this new number must be greater than the revision that was checked out.

Revision Numbers: *Major* [. *Minor*]*

☞ Revision numbers of the form 3.0 are not allowed.

To specify a particular revision of a file append a comma followed by the desired revision number to the end of the name, i.e. file.c,3 refers to revision 3 of file.c (commas are not allowed in project file names). The first command in the following example checks out the latest (current) revision of file.c. The second command checks out revision 3 of file.c regardless of what the current revision is:

```
CheckOut file.c
CheckOut file.c,3
```

The following command checks in file.c changing the revision to 4.1: (Note: this is only legal if the revision that was checked out was less than 4.1, e.g 4, 3.9, 4.0.9, or 2 etc.)

```
CheckIn file.c,4.1
```

In addition to supporting a sequence of revisions to a file in a project, Projector also allows users to create branches, alternative sequences of revisions that are parallel to the main revision sequence. In figure 3 revisions 1, 2, 3, and 4 form the main **trunk** of file.c's revision tree. Revisions not on the main trunk form branches. These branches can be easily identified by the alphabetic character embedded in the revision number. For example, the user can checkout revision 2 of a file and check it back in as revision 2a1, instead of revision 3. This begins the new sequence, 2a1, 2a2, 2a3, etc. A second branch off revision 2 would create revision 2b1. Revisions off branches follow the same default numbering scheme as revision on the main trunk, i.e. 1, 2, 3, etc. However, the user can use major/minor numbering, with an arbitrary number of minor components.
When specifying a revision, a name such as "file.c,2a" implies the latest revision on the "a" branch off revision 2. If there are two revisions, 2a1, and 2a2, then the revision 2a2 will be used.

To refer to particular revisions when using The "Check Out" window the user can double click on a file to display its revision tree. The individual revisions can then be selected and acted upon, such as checking out a particular revision of a file, or getting information about that revision.

User Names

Most Projector commands requires a user name in order to keep track of who did what. Projector has reserved the shell variable "User" as a place to maintain the current user. When using Projector interactively, via its windows, the current value of {User} appears in the "User:" field. On the command line there are two ways to specify the current user; the order of precedence (from greatest to least) is:

1. Use the name given on the command line (via the "-u" option)
2. Use the name given in the {User} variable. The {User} variable is a predefined variable that the MPW Shell initialized at launch time. It is initialized to the value in the User Name field in the Chooser.

If a name cannot be determined an error is reported and the command aborts. The above description for determining the user applies to most Projector commands. Any exceptions are noted on the command page for the appropriate command.

☞ User privileges should be handled by AppleShare. Since AppleShare determines privileges when a network volume is initially mounted changing the {User} variable will not change the access privileges to those corresponding to the new user.

Symbolic Names

Projector supports a general purpose naming facility that allows project users to easily identify files, revisions and branches within a project. The first character of a symbolic name (or 'Name') cannot be a digit (0-9). Also, commas, greater than or less than symbols ('<', '>'), and dashes ('-') are not allowed anywhere in a Name. Names are kept on a per project basis, and can refer to at most one revision per file in that project. A Name can be used anywhere a list of files can be used, and finally, names are not case sensitive. For example, the following commands check out three files.

```
NameRevisions Work file.c file.h library.c
CheckOut Work
```

☞ Projector needed its own naming facility rather then using the Set command and the existing Shell variable mechanism for the following reasons:

- Names can only refer to one revision per file. Shell variables are arbitrary text macros so this restriction could not be enforced.
- Names are kept on a per project basis. In Projector the "scope" is the current project. In the Shell, scope is based on nested command files.
- Names do not need funny delimiters ('{' and '}') in order to be recognized.

Of couse you can use shell variables if you'd like.

By default, Names are expanded to the revision level when they are used, not when they are defined. In the above example the Name "Work" will expand to the latest revisions of the three files each time "Work" is used. This means the revisions that "Work" implies will change as new revisions to those files are created. To explicitly bind a revision to a Name the revision number must be included at the time of definition. The following example illustrates the differences:

```
NameRevisions Work file.c,4 file.h,3 library.c
```

The Name "Work" will expand to revision 4 of file.c and revision 3 of file.h. However, library.c will always expand to the latest revision. The "-e" option will expand all files to the revision level during definition, for example:

```
NameRevisions -e Work file.c file.h library.c
```

This is equivalent to:

```
NameRevisions Work file.c,6 file.h,3.5 library.c,7
```

Where the specified revisions are the latest revisions of the respective files. The "-e" option saves the trouble of determining the latest revision of each file.

Names are recursively expanded until no further expansion can occur or a comma is found. For example, given the following Names:

```
NameRevisions defs.h defs.h,1.1
NameRevisions file.c file.c,2
NameRevisions Work file.c defs.h,2.1 library.c
```

The following CheckOut command:

```
CheckOut Work
```

Expands to:

```
CheckOut file.c,2 defs.h,2.1 library.c
```

Since an explicit revision was specified for "defs.h" in the definition of "Work" the expansion of "defs.h" to "defs.h,1.1" did not occur.

Names are particularly useful when working on a branch of a file. For example, suppose a programmer is designing a new algorithm in file.c and wants to implement the algorithm on branch 4a of file.c. By defining the following Name:

```
NameRevisions file.c file.c,4a
```

the programmer can automatically check out and check in the latest revisions on the 4a branch.

```
CheckOut -m file.c
```

The above command will check out a modifiable copy of the latest revision on the 4a branch of file.c. The user can override the Name, simply by specifying a particular revision along with the name.

```
CheckOut file.c,3
```

This will check out revision 3 of file.c, regardless of any Names. Because an explicit revision was given no Name expansion occurs. A comma with no subsequent revision number implies the latest revision on the main trunk of the file:

```
CheckOut file.c,
```

This will check out the latest revision on the main trunk of file.c. If "file.c" had not been defined as a Name (see a few examples above) then the comma at the end would not be necessary.

Names can be defined recursively in a project tree. Going back to figure 1 as an example, suppose Bob wanted to "freeze" the current state of his projects and name the current version "Release 1":

```
NameRevisions -e -a -r -project Tools "Release 1"
```

This would create a name "Release 1" in each of the projects that would expand to the latest revisions as of when the name was defined. The above command is equivalent to the following:

```
NameRevisions -e -project Tools∫ "Release 1" -a
NameRevisions -e -project Tools∫Sort "Release 1" -a
NameRevisions -e -project Tools∫Rez "Release 1" -a
NameRevisions -e -project Tools∫Count "Release 1" -a
```

It is very important to understand the difference between the above commands and the following command (notice that the "-e" option is missing):

```
NameRevisions -r -project Tools "Fred" -a
```

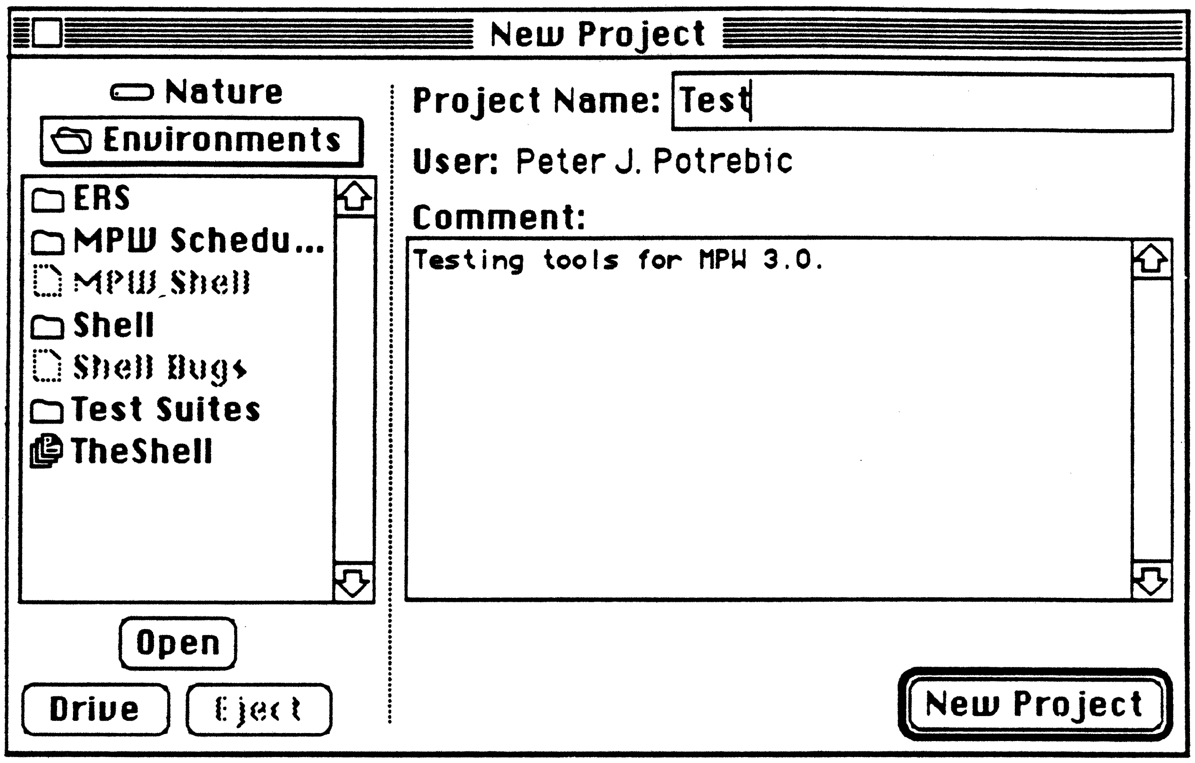
The Name "Fred" will be expanded to the latest revisions each time it is used. The Name "Release 1" will always expand to the latest revisions that existed when the name was defined.

Both public and private (the default) Names are supported. Public Names are visible to all members of the project. Private Names are only visible to the individual who created them, and can be declared in the UserStartup file using the NameRevisions command. Public Names are stored in the project itself.

Working on a Project

Project Creation

The simplest way to create a project is to create it interactively using the "New Project" window (see figure 4). The window can be displayed by using the -w option to NewProject. The other Projector windows (Check In and Check Out) can be displayed in a similar fashion. Once the windows are visible the standard Macintosh windowing techniques apply.



Project Name: Test

User: Peter J. Potrebic

Comment:

Testing tools for MPW 3.0.

Figure 4 - "New Project" window.

The "New Project" window is fairly self explanatory. The left most pane of the window is a standard file like control where the HFS file structure is shown. The difference between this control and standard file is that projects are graphically indicated by a small icon representing a project (see figure 4). This allows the user to create a project anywhere in the file system, either under an existing project or in some other directory. In figure 4 the directory "TheShell" is listed and it happens to be the project directory (represented by the multiple document icon) for the "TheShell" project.

To create a new project via the command line requires a single two word command:

```
NewProject Test
```

This creates a project named *Test* whose project directory, created by Projector, is *:Test:*. Projector will maintain all information regarding this project in the project file within this directory. Nested projects will appears as folders within this directory. The checkout directory is set to the current directory at the time of the check out. Every user can have a different checkout directory, and this directory can be set using the CheckOutDir command.

Test also becomes the current project and is automatically mounted for you. If the new project does not have a parent project, a MountProject command should be added to your UserStartup file, followed by a CheckOutDir command to specify where the files to be checked out should go.

Test can actually be an HFS pathname or a Project pathname. In either case the project name is the leaf of the path. If an HFS path is given, that directory becomes the project directory for the new project. If a project pathname is given the new project becomes a subproject of its parent. For example:

```
NewProject MPW∫Tools∫Fortran
```

This creates a new project Fortran that is a subproject of the Tools project. If the project directory of the Tools project is FS:Projects:MPW:Tools then the project directory of the Fortran project would be FS:Projects:MPW:Tools:Fortran.

The following example is equivalent to the previous example:

```
NewProject FS:Projects:MPW:Tools:Fortran
```

Since the project directory of the Tools project is FS:Projects:MPW:Tools the Fortran project automatically becomes a subproject of the Tools project.

When creating a project a descriptive comment can also be given. This is useful so no one will forget why they keep coming to work every morning – if they forget they can look at this comment to refresh their memory (see the ProjectInfo command).

Locating Projects

The set of mounted projects defines a set of project trees. This tells Projector both the names of the mounted projects and where their project directories are located. If a project is not in one of those trees the project cannot be accessed. If a project is moved or renamed (changing its project directory) users must change their MountProject commands in order to re-connect to the project.

Checkout and CheckIn

The simplest way to check out a file is to use the "Check Out" window (see figure 5). When browsing through the project hierarchy in this, or any other Projector window, the following visual cues are used to convey file ownership:

□	file is free, no own has the last revision checked out for modification
✎	the current user has the latest revision of this file checked out for modification
🔒	some other user has the latest revision of this file checked out for modification

☞ A checked out file matches its corresponding checked in file in all ways except for the 'ckid' resource that Projector places in every file in the project to correctly identify copies that the user has checked out..

The following description refers to figure 5.

The two radio buttons at the bottom of the of the window specify read-only or write-modify check out. The default is to check out read-only copies.

The "Checkout to:" field is a pop-up menu that allows the user to pick the checkout directory (default), the current directory, or any other directory. Checked out files will be placed in the this directory which defaults to the checkout directory.

The "Select all" button will select all files whose most recent revisions are not checked out for modification.

If the user is doing a read-only check out, the "Select newer" button will select all the files the user doesn't already have checked out for modification by comparing each file in the checkout directory with the latest revision of that file in the project. This is a convenient way of checking out the "latest" revisions in the project.

☞ The two buttons "Select all" and "Select newer" do not actually check out files, they simply make a selection in the Project list. Only the "CheckOut" button (the button in the lower right-hand corner of figure 5) actually checks out the selected files.

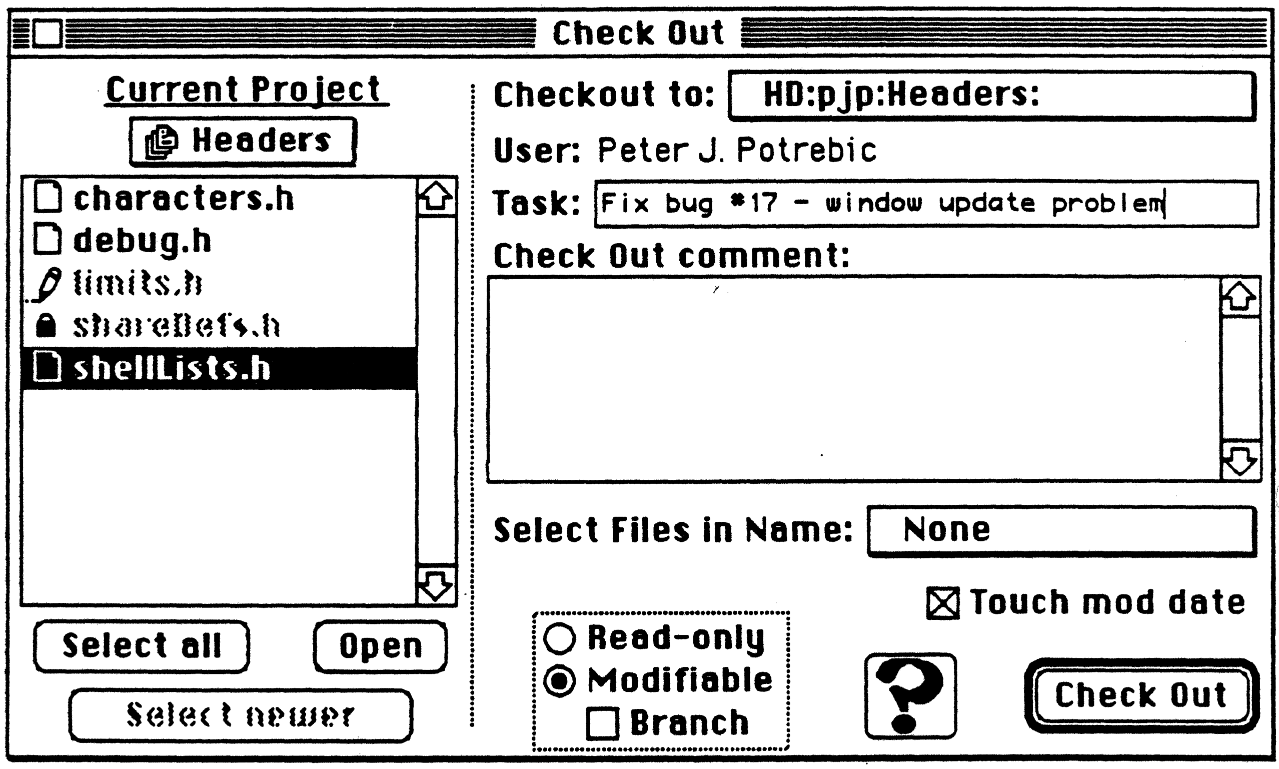


Figure 5 - "Check Out" window.

When checking out a file for modification, the "Task" and "Comment" fields allow the user to specify the purpose of the check out. This allows other users to determine the reason the file(s) were checked out. Both fields are saved in the 'ckid' resource of file, and Projector allows the owner to edit these fields in the Check In window.

In normal circumstances it is bad practice to check out a writeable copy of a file that already is checked out for modification. To prevent this from happening Projector does not allow users to select files that are already locked (names are dimmed). Since the default is to check out the most recent revision on the main trunk, a file is considered locked when the most recent revision is checked out for modification. However, checking the "Branch" control will allow selection of locked files. Checking out a locked revision will automatically create a branch off of that revision.

Users can check out a particular revision of a file by displaying the revision tree (double click on the file) and selecting the revision they want.

Files can also be checked out using the "CheckOut" command:

```
CheckOut file.c -m
```

This will place a modifiable copy of file.c in the checkout directory of the current project. The checkout directory can be changed using the CheckOutDir command. If no project has been mounted or if file.c does not exist in the current project an error is reported. Along with giving the user a copy of the file several pieces of information will be saved in 'ckid' resource of the file's resource fork:

- The project that the file came from.
- The name of the file itself.
- The revision of the file that was checked out.
- Whether the file is a read-only, modifiable, or modified read-only copy.
- The user that checked out this copy of the file.
- Date and time of the checkout.
- The Task.
- The Comment.

When a file is being checked out the default action is to place the file in the checkout directory for the project. However, a different directory can be used. The rules for the determining the directory are as follows, from highest to lowest precedence:

1. The directory indicated if a non-leaf name is specified.
2. The directory specified with the "-d" option.
3. The checkout directory for the project (see CheckOutDir command).

For example:

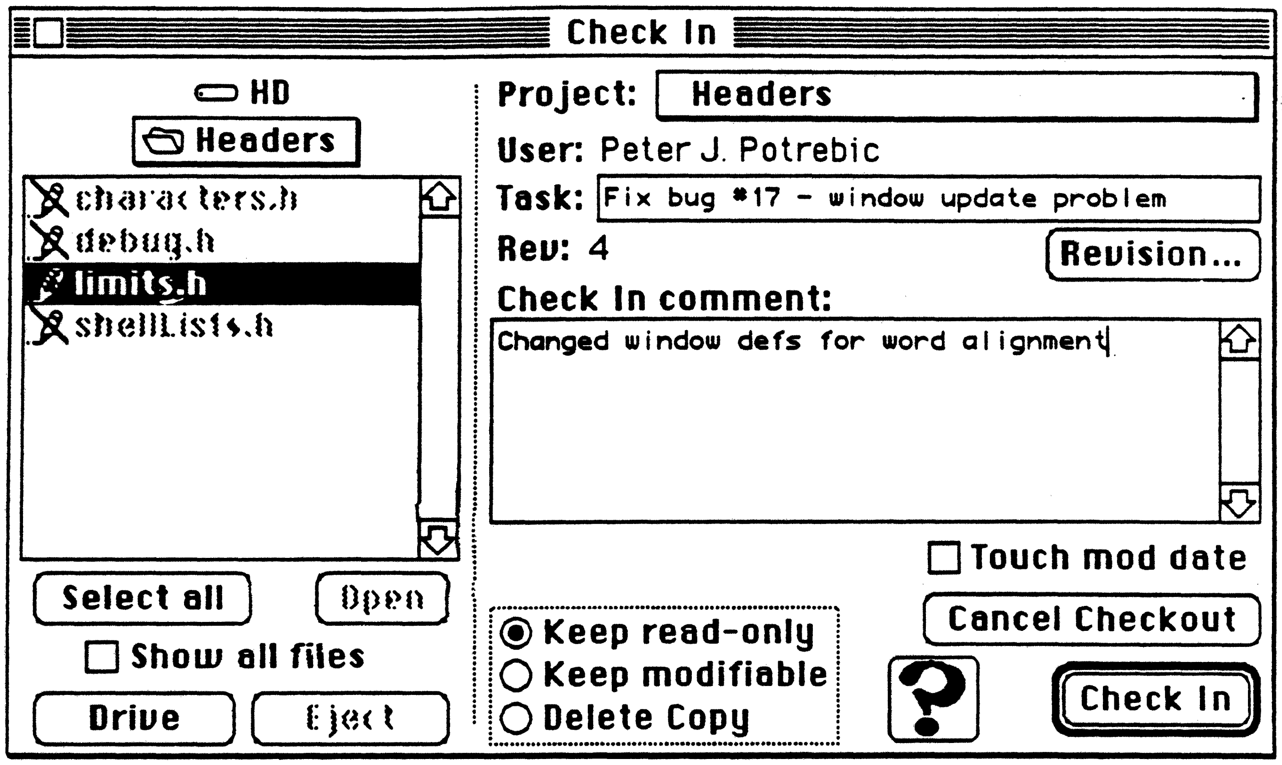
```
CheckOut -d hd:MPW: file.c hd:work:defines.h
CheckOut hd:MPW:main.c library.h
```

This first CheckOut will place a copy of file.c in hd:MPW:file.c and a copy of defines.h in hd:work:defines.h. In this case the checkout directory was not used. The second CheckOut will place a copy of main.c in hd:MPW:main.c and a copy of library.h in the checkout directory for the {Project} project.

CheckIn is used to create new revisions to files, implying that changes have been made and the user wants to add the changes to the project. This means that checking in read-only files does not make sense since read-only copies do not contain changes and therefore cannot create new revisions. When the changes to modifiable copies are complete, the CheckIn command will submit the changes and create a new revision of the file:

```
CheckIn file.c
```

The "Check In" window can be used (see figures 6) for the same purpose.



Project: Headers
User: Peter J. Potrebic
Task: Fix bug #17 - window update problem
Rev: 4
Check In comment:
Changed window defs for word alignment

[✓ Keep read-only]
Keep modifiable
Delete Copy

Figure 6 - "Check In" window

When checking out a file the project to which the file belongs needs to be specified. Since this project is "remembered" in the resource fork of the file it does not have to be specified on check in. All Projector needs to know is what file to check in. When using the CheckIn command, files belonging to different projects may be specified at the same time.

In the "Check In" window (figure 6) the current project can be selected with a pop-up menu in the "Project:" control. Only the files that belong to the current project are displayed in the standard file control.

☞ If, for whatever reason, the 'ckid' resource of the file is corrupted or removed then Projector cannot identify the file and it becomes an **orphaned file**, and no longer belongs to any project. If you still need to check the file in, then move or rename your copy, check the file out again (you may need to cancel the checkout using the "Cancel Checkout" button in the "Check In" window) and use the transferCkid command as documented in its command page.

In the "Check In" window selecting a file that is not currently checked out is not allowed. That is, only the file names in boldface can be selected. This is the restriction saying that only files that have been checked out for modification can be checked in.

The standard action after checking in a file is to leave the user a read-only copy of the file. The radio buttons at the bottom of the "CheckIn" window have this and two other choices. The "Keep read-only" radio button keeps a read-only copy in your directory. The "Keep modifiable" radio button lets you check the file in and still retain a modifiable copy. The "Delete Copy" radio button deletes your copy of the file once it is successfully checked in. These last two radio buttons correspond to the "-m" and "-del" options of the CheckIn command.

To throw away any changes use the "Cancel Checkout" button located in the lower right-hand corner of the windows - above to the "CheckIn" button. This is especially useful when if half the file was accidently deleted or you goofed and checked out a file for modification when you really wanted a read-only copy.

The "select all" button selects all the files checked out for modification by the current user in the directory listed in the Standard File list. The selected files can then be checked in by clicking the "Check in" button.

New files can be added to projects with the "-new" option to the CheckIn command, or by checking the "Show all files" control in the "Check In" window. When that button is checked all files in the current directory are shown. Files not belonging to any project can be selected and checked in. The files will be added to the current project.

Below is a list and a short description of all the different icons that can appear in the Check In window:

✎	modifiable file from the current project owned by current user	
✗	Read-only file from current project	
✎	modified read-only file from current project	
🔒	modifiable file from current project, owned by another user	
☐	file not belonging to any project	Only appear if "show all files" is selected
⍰	modifiable file from another project	
⍰	read-only file from another project	
⊘	file with a corrupt or out-of-date "ckid" resource	

Branching

A branch can be created during the checkout or checkin process. Checking out a modifiable copy of an old revision automatically creates a new branch (see figure 7). When file.c is checked back in it will automatically become revision 2a1.

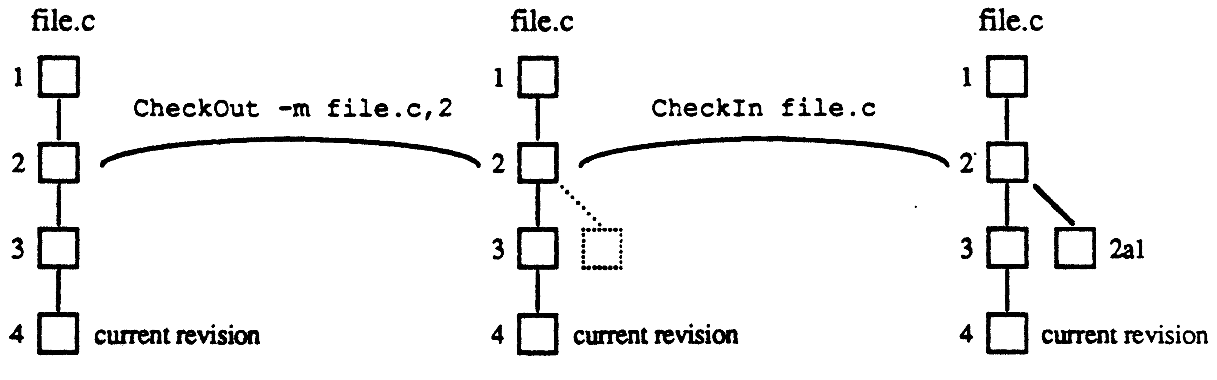


Figure 7 - A changing revision tree.

The following command will create a branch when checking in a file:

```
CheckIn main.c -b
```

In the above example the user did not need to specify a revision number in order to create a branch. The branch is automatically created off the revision that was checked out. When a file is checked out Projector remembers the revision that was checked out. When a file (obtained from revision *x*) is checked back in it can create a revision in one of two places:

- The next revision after *x*, continuing on the same line.
- On a branch off revision *x*.

Looking at figure 7 the user could not check in file.c as revision 3a1 or revision 6.

The following command will create a branch and number the first revision 1:

```
CheckIn main.c,1 -b
```

If revision 4 of main.c was initially checked out the above CheckIn command would create revision 4a1 (or 4b1 if revision 4 already had one branch, etc.).

Merging Branches

The initial release of Projector will not support automatic merging of files. The user can manually merge two branches (revisions) by checking out one revision as a read-only copy into a temporary directory and checking out the second revision for modification into a different directory (e.g. the checkout directory). The user can then use the Compare tool to find the differences and manually cut and paste the changes into the file checked out for modification. This file can then be checked in and the read-only copy in the temporary directory can be discarded.

Project Administration

The administrative duties for projects under Projector are very simple. Anyone who has write access to the project (under AppleShare) can administer the project. Responsibilities include:

- deleting old revisions of files that are no longer needed. This can be done to reduce the size of the project on disk.
- moving and renaming projects.
- Keeping track of the project log to ensure that the project is not being abused.

Moving & Renaming Projects

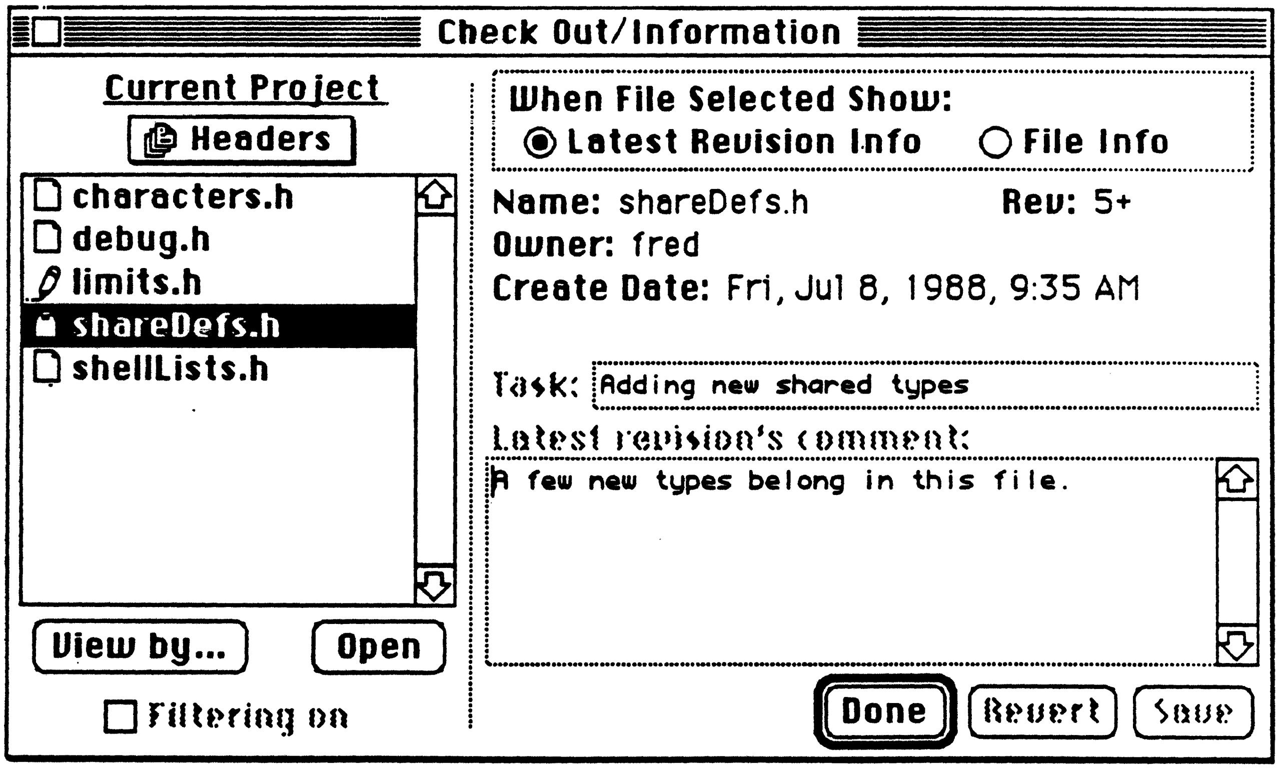
Projector was designed in such a way that a project could be moved or renamed using the Finder or the regular MPW commands. However, there are a few areas of concern. First of all, when a project is moved or renamed the project hierarchy is changed; users of this project must update their root project list to reflect the changes. Secondly, projects can be moved or renamed only when there are no files checked out for modification, and that after the Project has been changed *all* read-only copies be checked out again. This is because Projector puts the full-path project name in the resource fork's of files during checkout; once the project is moved or renamed the information is no longer valid.

☞ No other Finder or MPW operations are allowed on project directories.

Project Information

Information retrieval is one of the most important aspects of any source control system. There are several different ways to get information out of Projector: via the "ProjectInfo" command (see the ProjectInfo command), or via the "CheckOut" and "CheckIn" windows using the Question mark button (see figures 8-11). The information that can be retrieved from the project includes:

- **Project Information**
  - author - person responsible for the project
  - last modification date of the project
  - project comment
  - project log
- **File information**
  - author - person responsible for the file
  - last modification date of the file
  - file comment
- **Revision Information**
  - author
  - task
  - date the revision was created
  - revision comment



When File Selected Show: [✓ Latest Revision Info] File Info

Name: shareDefs.h Rev: 5+

Owner: fred

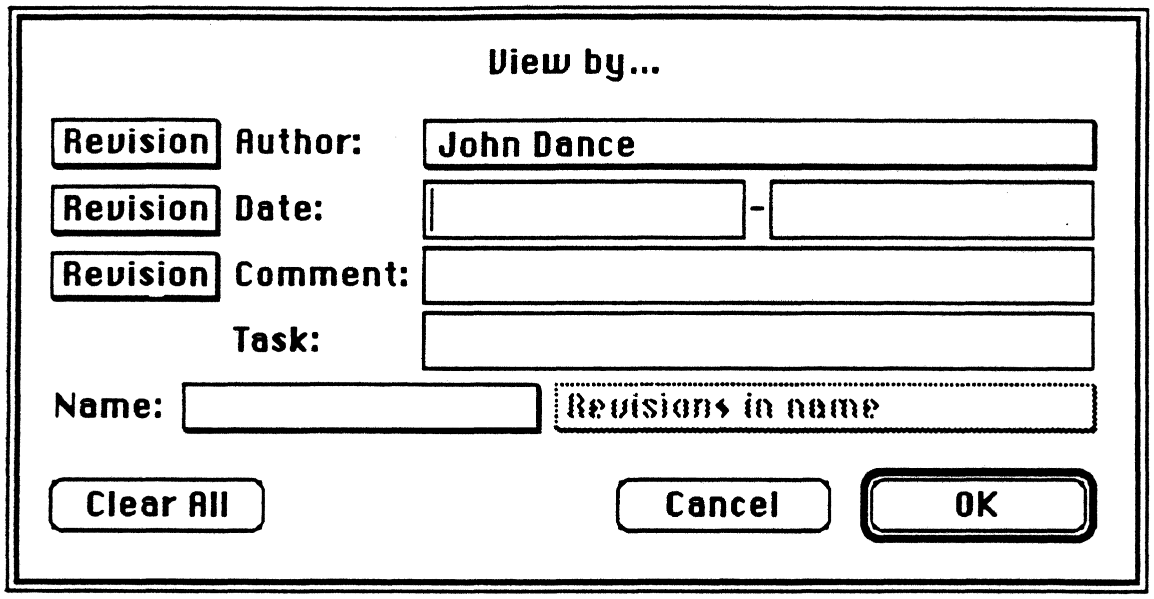
Create Date: Fri, Jul 8, 1988, 9:35 AM

Task: Adding new shared types

Latest revision's comment:

A few new types belong in this file.

Figure 8 - Project information in the "CheckOut" window.


View by...

Revision Author: John Dance

Revision Date: -

Revision Comment:

Task:

Name: Revisions in name

Figure 9 - View by... filter.

The "CheckOut" window's information (see figures 8 thru 9) is oriented toward browsing through the project to obtain information about individual files and revisions. The command line interface can handle more complex batch type requests such as: list all the revisions, including comments, that Bob made to file.c of the Sort project.

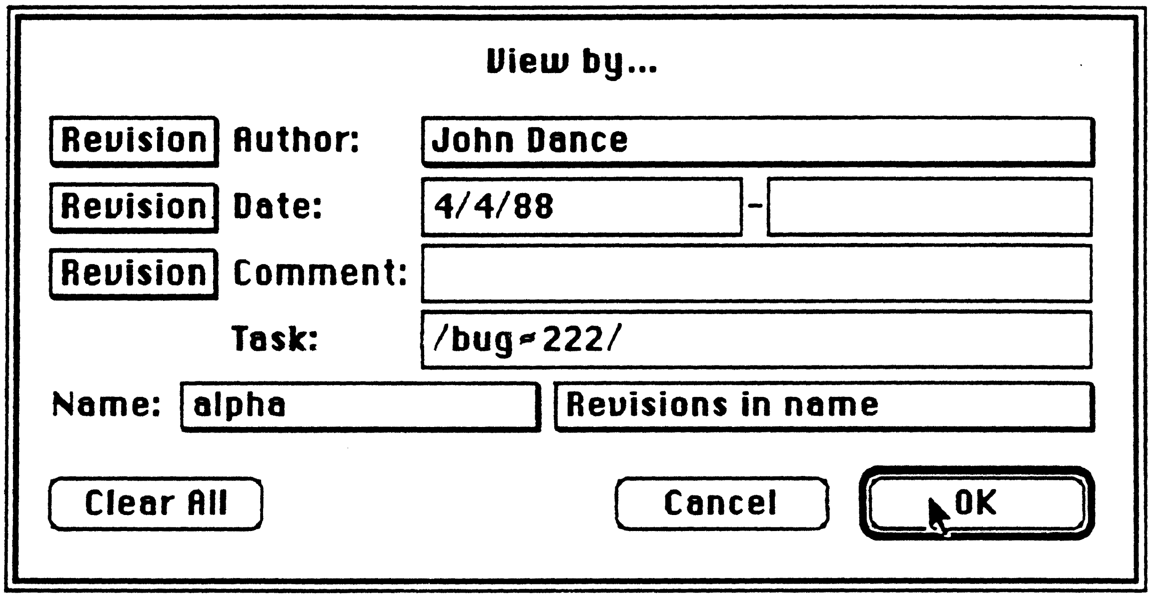
This can also be done with the "CheckOut" window, by selecting a subset of the project to view via the "View by..." dialog. The "View by..." dialog provides different items with which you may filter the files or revisions displayed in the list. Only files or revisions that match the criteria you have chosen will be displayed. To specify a filter, bring up the "View by..." dialog, and select the different items that are important to you. You may specify the following items:

- The author of a file or revision. All the authors known to the project will be listed in a popup menu. Select the desired author from the list.
- The file modification date or revision creation date. Type in the starting and ending dates. The format is dd/mm/yy [hh:mm[:ss] [AM|PM]]. If you would like to specify "on or since a date" enter the starting date in the first box, and leave the second box empty. If you would like to specify "before or on a date" enter the ending date in the second box, and leave the first box empty.
- File or revision comments. Type in either a literal string, or a regular expression in slashes (/regular expression/).
- Task comments. Type in either a literal string, or a regular expression in slashes (/regular expression/).
- Name. The popup menu will contain all your private names followed by the project's public names. Select the desired name from the list. You may also specify a relation to that name. (For example, to list all the revisions since "Alpha".) Select the desired relation from the popup next to the Name.

For the author, date, and comment items, you will need to specify if it should be applied to files or revisions.

☞ If you have specified a filter, and all the files or revisions are still being displayed, check the revision/file setting on your selection criteria.

For example, in Figure 10, the user has specified a filter to list all revisions in "alpha", created by John Dance, on or after April 4, 1988, dealing with Bug #222.


View by...

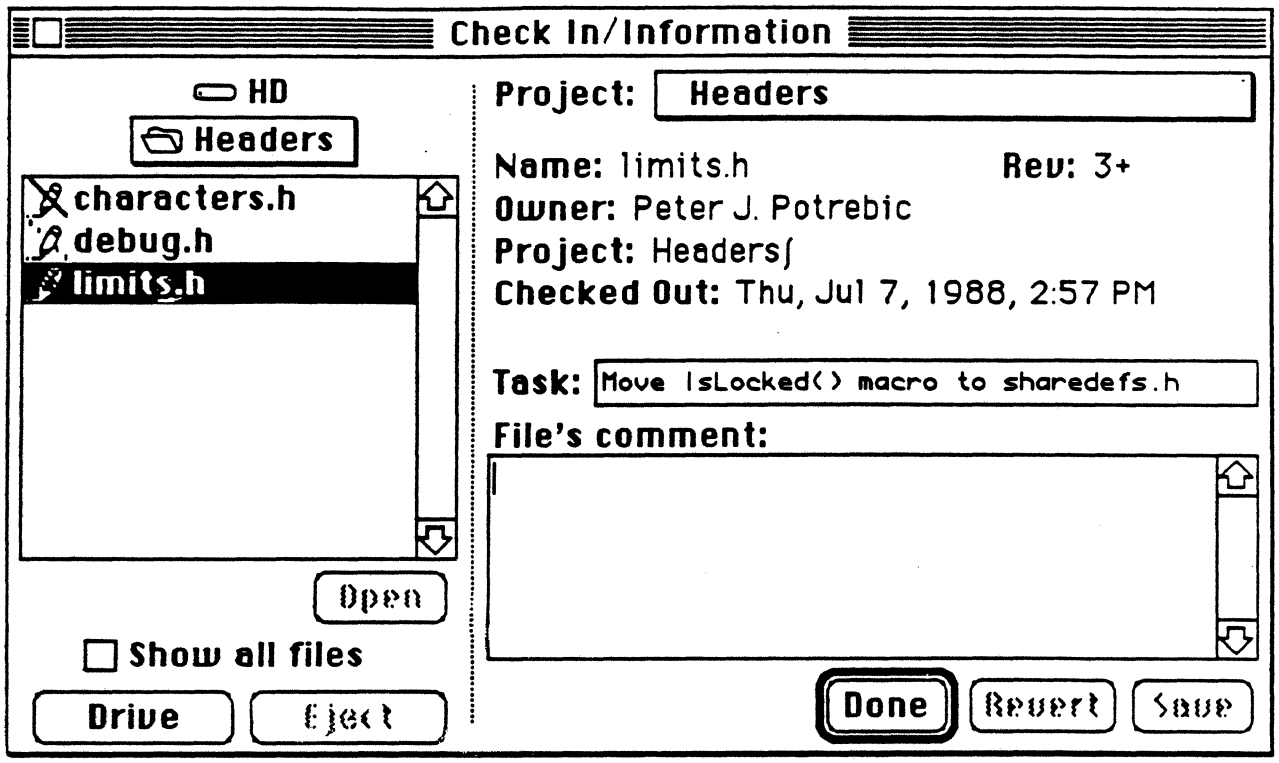
Revision	Author:	John Dance	
Revision	Date:	4/4/88	-
Revision	Comment:		
	Task:	/bug≈222/	

Name: alpha — Revisions in name

Figure 10 - View by... dialog with selection criteria

Selecting a project will display the project information. Selecting a file will either display the current state of the file, that is the status of the latest revision or it will display the file information. Which is displayed depends on the radio buttons at the top of the window (see figure 8). Double clicking on a file will display its revision tree. The latest revision will be selected by default, with its information (status) displayed. Selecting another revision will display its status. The comment and task fields are editable so changes or additions can be made.

☞ Deleting comments is not nice! Use Projector to record history - not destroy it.



Project: Headers

Name: limits.h
Rev: 3+
Owner: Peter J. Potrebic
Project: Headers∫
Checked Out: Thu, Jul 7, 1988, 2:57 PM

Task: Move IsLocked() macro to sharedefs.h

File's comment:

Figure 11 - Project information in the "CheckIn" window.

Appendix

'ckid' Resource

The following describes the 'ckid' resource that Projector maintains in the resource fork of all files that belong to a project.

Resource Name	'ckid'	CheckOut IDentification.
Resource Contents	Project name	full project pathname of the project
	File name	name of the file
	Revision number	the revision that was checked out
	State	read-only, modified read-only or modifiable
	User name	name of user who checked out the file
	Date-Time	date and time of the checkout
	Task	contents of task field when checked out
	Comment	contents of comment field when checked out

Appendix

The Project Directory

A project resides in an HFS directory called the project directory. The name of this directory is the name of the project. Users need not worry about what goes on inside the project directory, and they are warned not to place their own files in these directories. The following is just given for completeness.

Project file

The entire project, including all the files, revisions, comments, etc is kept in a single HFS file called "ProjectorDB" with the type 'MPSP'.

Appendix

Glossary of Terms

Author	With respect to revision it is the name of the person who made a revision. For files and projects it is the person with primary responsibility for that file or project.
Branch	An alternate sequence of revisions emanating from another revision and running parallel to the main trunk.
CheckOut Directory	This is the directory where, by default, Projector will place checked out files. Each project has a corresponding CheckOut directory which can be changed with the "CheckOutDir" command.
'ckid' resource	A resource that Projector maintains in the resource fork of all files belonging to a project in order for identification purposes.
Comment	Text describing the revision, file, or project.
Current Project	The name of the current project. Projector assumes all actions pertain to this project unless a different project is specified with the "-project" option.
File Information	Information maintained by Projector on a per file basis. Includes: • Author • Last modification date • Comment
Locked Revision	A revision that is currently checked out for modification.
Mounted Project	A project that is not nested beneath another project. Similar to the root directory on a volume. A user can mount several projects, just as they may mount several volumes. All projects under the mounted project can be accessed by the user.
Name	An identifier that represents a set of files, revisions and branches, with the restriction that a name can only refer to one revision in any one file.
Orphaned file	A file that belongs to a project, but it resource fork no longer contains the information that Projector needs to determine to which project it belongs.
Project	A set of files and zero or more projects.
Project Directory	The directory where Projector maintains all the information with respect to a given project.

Project File	The file (always named ProjectorDB) where an entire project is maintained. There is one and only one project file within every project directory.
Project Information	Information maintained by Projector on a per project basis. Includes: • Author • Last modification date • Comment
Project Log	The log that records all actions that delete information from the project. Can be printed oput using the -log option to projectInfo.
Project Name	The name of the project also the name of the directory containing the project.
Revision	An instance of a file in project. A new revision is created each time a file is checked in.
Revision Information	Information maintained by Projector on a per revision basis. Also known as the current state of a revision. For unlocked revisions this includes: • Author • Creation date • Comment • Task For locked revisions the information is: • Author (person who checked out the file) • Check out date • Task
Revision Number	A unique number identifying a revision within a file.
Revision Tree	The composite history of a file, that is all the revisions and branches made to a file. The revision tree for a file can be displayed via the Status command or by double clicking a file name in the Project hierarchy pane.
Task	A short description of the task the person accomplished with a revision.
{Task}	The name of the current task. It appears in the "Check Out" and "Check In" windows as the default task.
Trunk	The main sequence of revisions to a file.
User	Each project has one or more users that are permitted to access files in a project.
{User}	The name of the current user. Projector logs this name with all transactions. This can be overridden by specifying a different name with the "-u" option available in all Projector commands.

Wish-List for Future Releases

Naming Revisions

The initial release of Projector will only support revision naming of the form:

fileName,revNum

where the RevNum may include branches. Future releases will support of more powerful naming mechanism where revisions could be named by date/time, or by a symbolic name:

```
CheckOut file.c,9/8/87
CheckOut file.c,Beta1
```

The first command would checkout *the* latest revision of file.c that existed on the given day (at 12pm). The second command would checkout the Beta1 revision of file.c, assuming the file.c has a Beta1 revision. A further extension would be to include ranges in order to specify a set of revisions:

```
ProjectInfo file.c,Beta1-Beta2
```

This command would list all the revisions that were created between the Beta1 revision and the Beta2 revision of file.c. Dates and times could also be used in place of or along with the names.

Preferences

Configurable preferences would be a useful enhancement. These preferences could be set using the Set command. Commands could be added to the startup file in order to automatically configure the preferences every time MPW is launched. Possible preferences include:

- Open window on checkout (only applies to text files).
  - Open all files checked out.
  - Only open files checked out for modification.
- Maintain menu of checked out files.
  - Display all files checked out.
  - Only display files checked out for modification.
- Multiple owners not allowed.

Additional Windows

Projector could be made easier to use with the addition of several more windows:

"Name Revisions" Window
A window would allow users to interactive create new Names and modify existing Names.

"Merge Revisions" Window
An interactive merge process could facilitate resolving conflicts that arise when several users make modifications to the same file. In this window a user could

interactively merge the files with Projector's help. When Projector detects a conflict the associated text could be so marked. The user could then take the necessary steps to resolve the conflict.

"Compare Revisions" Window

This window would be very similar to the MergeRevisions window.

Project Path Names

The first release of Projector will only support full project path names. Future releases should support a naming mechanism that mimics HFS so that partial paths are allowed. Setting the current project would then be equivalent to changing directories. (For the following examples assume that the current project is set to "MPW".)

```
Project shell
```

The above command would then set the current project to MPW∫shell, that is the shell project within the MPW project. The first release of Projector only allows commands like the following:

```
Project MPW∫shell
```

Merging Branches

Revisions can either be merged automatically, or the user can get a copy of the revisions, merge them himself, and then check in the merged file. Using Figure 8 as an example the following command will merge revisions 2a1 and 4:

```
MergeRevisions file.c,2a1 file.c,4
```

This will give the user a modifiable copy (the target revision, 4, is implicitly checked out for modification) of the file containing the merged text. Or the user can check out a read-only copy of revision 2a1 and a modifiable copy of revision 4 and then merge the files into the copy of revision 4, either manually or with the Merge command. The merged file can then be checked in creating a new revision to file.c.

Part II—Commands

Projector Command Summary

CheckIn -w | -close

CheckIn [-u *user*] [-project *project*] [-t *task*] [-cs *comment* | -cf *file*] [-n | -y | -c]
[-del | -m] [-new | -b | -cancel] [-touch] (-a | *file*...)

CheckOut -w | -close

CheckOut [-u *user*] [-project *project*] [[-m | -b] [-t *task*] [-cs *comment* | -cf *file*]]
[-d *directory*] [-n | -y | -c] [-r] [-open] [-noTouch]
(-update | -newer | -a | *file*...)

CheckOutDir [-project *project*] | -m] [-r] [-x | *directory*]

DeleteNames [-u *user*] [-project *project*] [-public] [-r] [*names*... | -a]

DeleteRevisions [-u *user*] [-project *project*] [-checkout] *revision*...

ModifyReadOnly *file*

MountProject [-d] [*project*]

NameRevisions [-u *user*] [-project *project*] [-public | -b] [-e] [-r] [-s]
[*name* [*names*... | -a]]

NewProject -w | -close

NewProject [-u *user*] [-cs *comment* | -cf *file*] *project*

OrphanFiles *file*...

ProjectInfo [-p *project*] [-comments] [-revisions] [-f] [-r] [-s] [-only | -m]
[-af *author*] [-a *author*] [-df *dates*] [-d *dates*] [-cf *pattern*] [-c *pattern*]
[-t *pattern*] [-n *name*] [*object*...]

TransferCkid *sourceFile destinationFle*

Changes (7/5/88):

- Added the modifyReadOnly command
- **CheckIn:**
  New options: "-cancel" and "-touch"

- **CheckOut**:
  New options: "-update" and "-noTouch"

- **ProjectInfo**:
  New option: "-log"

Changes (4/14/88):

- This is the alpha release.

- "-p *project*" option was renamed to "-project *project*"

- **CheckIn**:
  "-n" option renamed to "-new"
  "-c" option renamed to "-cs"
  New options: "-close", "-y", "-n", "-c"
  "-d *directory*" option was removed. It was deemed unnecessary.

- **CheckOut**:
  "-c" option renamed to "-cs"
  New options: "-close", "-y", "-n", "-c"

- **DeleteNames**:
  New command

- **DeleteRevisions**:
  Deleted the "-n" and "-d" options (for names and dates)
  Added the "-checkout" option to delete check outs.
  Changed the semantics to delete all revisions (on the same branch) previous to the specfied revision.

- **NameRevisions**:
  New options: "-b", "-s"

- **Project**:
  New command

- **ProjectInfo**:
  "-ap *author*", "-dp *dates*", "-cp *string*", "-i", and "-w" options removed
  Changed syntax for specifying dates and names

Changes (1/25/88):

- Added the -w option to the CheckIn, CheckOut, NewProject, and ProjectInfo commands. This option brings up the respective window.

- Changed -w option in CheckOut to -m (modification)

- Changed default behaviour of CheckIn to leave a read-only copy of the file, rather then delete the user's copy.
- Various other options have new names.

CheckIn - check in files to a project

Syntax

CheckIn -w
CheckIn -close
CheckIn [-u *user*] [-project *project*] [-t *task*] [-touch] [-n | -y | -c]
    [-cs *comment* | -cf *file*] [-del | -m] [-new -b | -cancel] (-a | *file*...))

Description

Check the specified files back into the project creating a new revision. After check in the file will be a read-only copy of the newly created revisions.

Projector determines to which project the file belongs to looking for a Projector identification resource in the resource fork of the file. The identification resource is placed in the file during checkout. This allows files belonging to different projects to be checked in with a single command.

If the -a (all) option is used instead of *file*..., Projector checks in all of the files in the current directory that have been checked out for modification. The files are checked into their respective projects.

To add a new file to the project, use the "-new" option. The file will be added to the current project.

When the file is checked in, Projector automatically increments the revision number by one. For example, if revision 2.17 was checked out, the new revision will be 2.18. This default numbering scheme can be overriden by using the "filename,rev" notation. For example if file.c revision 2.17 was checked out, then the user could check it in as file.c,3 to jump to the next major revision level.

If no comment is specified (-cs option) then the comments that are saved with each file will be used. The same applies to the task.

Input

None.

Output

None.

Diagnostics

Errors and warnings are written to diagnostic output.

Status

The following status values are returned:

0	No Errors
1	Syntax Error
2	Error in Processing
3	System Error

Options

-u *user*	Name of the current user. This overrides the {User} shell variable..

-project *project*	Name of the project that contains the files. This project becomes the current project for this command.
-new	Add a new file to the current project.
-cs *comment*	A short description of what changes have been made to the file(s) being checked in. This comment will be associated with all the file(s) being checked in. This overrides any comment saved with the file.
-cf *file*	The comment is contained in the file *file*.
-t *task*	A very short description of the task that was accomplished by the changes made to the file(s). This overrides any task saved with the file.
-touch	Touch the modification date of the file after checking it into the project. This option does not do anything when used with the "-del" option.
-a	Check in all modifiable files in the current directory. The files will be checked in to their respective projects.
-b	Check the file in as a branch off of the revision that was checked out.
-cancel	Discard any changes made to the files being checked in. This is useful when the changes made should be thrown away.
-del	Delete the user's copy of the file after it is checked in.
-m	Keep a write-privileged copy of the file(s) for further modification. This basically check points the file(s), doing a checkin followed by a checkout for modification of the new revision.
-w	Open the "Check In" window. Bring it to the front if it is already open.
-close	Close the "Check In" window.
-y	Answer "yes" to any confirmation dialog that occurs, causing check in to proceed if any conflicts occur.
-n	Answer "no" to any confirmation dialog that occurs, skipping files that cause some type of conflict.
-c	Answer "cancel" to any confirmation dialog that occurs, causing check in to stop if any conflicts occur.

Examples

```
CheckIn file.c -cs "added some comments"
```

Checkin the file.c to the current project. A new revision of file.c will be created and the user will be left with a read-only copy of the file. The comment will be saved with the new revision. Since no revision number was specified, Projector will simply increase the revision number by one.

```
CheckIn file.c interface.c,5 -t "Added -x option" ∂
     -cf commentFile -del
```

This command will check in two files reading the comment from the file commentFile. The task will also be saved with the new revisions. The user's copies of the files will be deleted. The new revision for interface.c will be revision 5.

```
CheckIn file.c :main.c -m
```

This command will check in two files using the comments that are saved with each file (see the reference manual for further details on saving comments with checked out files). After the command executes the user will still have modifiable copies of the files. This shows how files can be check pointed, saving the changes to this point while allowing further modification to continue without needing to manually checkout the file.

```
CheckIn -m -cancel file.c
```

The above command would be used if the user wanted to throw away any changes and start over again. The "-cancel" option signals Projector to cancel the previous checkout and the "-m" option means that the user wants to check it out again. The result is that the user is given another modifiable copy of the file she originally checked out.

```
CheckIn -new file.c
```

To check a new file into the project use the "-new" option. The above command adds file.c the the current project.

```
CheckOut -project MPW∫Tools∫Sort file.c -m
...edit the file...
CheckIn -project MPW∫Tools∫Sort file.c -b
```

The above two commands illustrate the usefulness of the "-b" option. In this example the user checked out a modifiable copy of the latest revision of file.c in the Sort project, edited the file, and then, using the branch option, checked the file in as a branch off the revision that was initially checked out.

CheckOut - check out file revisions from a project

Syntax

```
CheckOut -w
Checkout -close
CheckOut [-u user] [-project project] [[-m | -b] [-t task] [-n | -y | -c]
          [-cs comment  | -cf file]] [-d directory] [-r] [-open] [-noTouch]
          (-update | -newer | -a | file...)
```

Description

Obtain copies of a particular revision of a file from the current project project; the default is to check out read-only copies. Unless otherwise specified, copies will be placed in the checkout directory associated with the project. The default behavior is to touch the modification date of the checked out files ensuring that builds are triggered (using the Make tool). This is especially important when getting read-only copies from the project.

If *file* is a leafname (e.g. file.c) then Projector will checkout the latest revision of the file from the current project . If *file* specifies a revision (e.g. file.c,22) then that revision is checked out.

If *file* is a partial or full HFS pathname (e.g. :work:file.c or HD:work:file.c), the file will be placed in the specified directory, overriding the checkout directory for the current proejct.

Finally, *file* may be a Name. See the NameRevisions command for more information names. The Name is expanded and the corresponding revisions are checked out.

The "-m" and "-b" options check out the specified revisions for modification. Projector marks that revisions as bing checked out and excepts that file to be checked back in at some point in the future. (Note: read-only files do not have to be checked back in, in fact they cannot be checked in.)

When checking out files for modification a comment and task can be specified to remind yourself and others why the files are being checked out. If no task is specified the contents of the {Task} variable will be used.

Input

None.

Output

None.

Diagnostics

Errors and warnings are written to diagnostic output.

Status

The following status values are returned:

0	No Errors
1	Syntax Error
2	Error in Processing
3	System Error

Options

-u *user*	Name of the current user. This overrides the {User} shell variable.
-project *project*	Name of the project that contains the files. This project becomes the current project for this command.
-d *directory*	The directory where the checked out files should go. This overrides the checkout directory for the current project. See the CheckOutDir command.
-t *task*	A very short description of the task to be accomplished by checking out files for modification.
-a	Check out all the files in the Project.
-m	Check out a modifiable copy of the file. This locks the revision preventing other users from inadvertently changing the revision.
-b	Branch. A modifiable copy of the file is checked out. When the file is checked back in it will create a branch off the revision that was checked out.
-cs *comment*	A short description of what changes have been made to the file(s) being checked in. This comment will be associated with all the file(s) being checked in.
-cf *file*	The comment is contained in the file *file*.
-update	Find all read-only files on the main trunk from the current project in the checkout directory (or the "-d" directory) and update them to the latest revision if they are older revisions. Files in the directory that have been checked out for modification are not affected. Files that are on branches are not affected. Files in the project, but not in the checkout directory are not checked out. This option cannot be used with the options to check out files for modification.
-newer	For all files in the project make sure that the lastest revision exists in the checkout directory (or the "-d" directory). Files in the checkout directory that have been checked out for modification or a on branches are not affected. This option cannot be used with the options to check out files for modification.
-r	Recursively execute the CheckOut command on the current project and all of its subprojects.
-open	Open the file after it is checked out. This option only works for files of type TEXT.
-w	Open the "Check Out" window. Bring it to the front if it is already open.
-noTouch	Do not touch the modification date of the checked out files.

-close	Close the "Check Out" window.
-y	Answer "yes" to any confirmation dialog that occurs, causing check out to proceed if any conflicts occur.
-n	Answer "no" to any confirmation dialog that occurs, skipping files that cause some type of conflict.
-c	Answer "cancel" to any confirmation dialog that occurs, causing check out to stop if any conflicts occur.

Examples

```
CheckOut -m -project MPW∫Tools∫Count file.c
```

Checks out a modifiable copy of the latest revision of file.c from the "MPW∫Tools∫Count" project. The file is placed in the checkout directory for the project.

```
CheckOut -project MPW∫Tools∫Count file.c,22
```

The above command checks out a read-only copy of revision 22 of file.c from the "MPW∫Tools∫Count" project. The file is placed in the checkout directory for the project.

```
CheckOut file.c -t "Fix Bug 7" -m -d "{MPW}ToolsSrc:Count"
```

This command will check out a modifiable copy of file.c. By setting the task other users will be able to see why this user has checked out file.c. The files are placed in {MPW}ToolsSrc:Count.

```
CheckOut -a -d HD:Work:Count
```

The above example checks out read-only copies of all of the files in the current project and places the copies in the directory HD:work:count.

```
CheckOut -a -project MPW∫ -r
```

Checks out read-only copies of all of the files in the MPW project and all of its subprojects. Its behavior is the same as if the user executed the following commands individually:

```
CheckOut -a -project MPW∫
CheckOut -a -project MPW∫Shell
CheckOut -a -project MPW∫Tools
CheckOut -a -project MPW∫Tools∫Sort
...
```

To conveniently update the read-only files (from the current project) without affecting any files checked out for modification use the "-newer" option:

```
CheckOut -update
```

Projector scans through the checkout directory of the current project and finds all the read-only files that are out of date (i.e. they aren't the latest revision on the main trunk). All such files are checked out.

See Also CheckOutDir

CheckOutDir - set checkout directory

Syntax CheckOutDir [-project *project*] | -m] [-r] [-x | *directory*]

Description Change the checkout directory associated with the current project to the HFS pathname *directory*. From this point on, files checked out of the named project will be placed, by default, into this directory. When a new project is created, the checkout directory is set ":", i.e. the current directory.

It is recommended that you put CheckOutDir commands for projects in UserStartup following the MountProject commands. This will automatically configure the Projector environment each time MPW is launched.

If *directory* is missing the checkout directory of the current project is written to standard output in the form of a CheckOutDir command.

Note that this command has no -u (user) option. Since the checkout directories are part of the MPW environment, which is not partitioned on a per user basis, a user name is not required.

Input None.

Output If *directory* is missing the checkout directory of the current project is listed in the form of a CheckOutDir command.

Diagnostics Errors and warnings are written to diagnostic output.

Status

0	No Errors
1	Syntax Error
2	Error in Processing
3	System Error

Options

-project *project*	Name of the project to associate the checkout directory with. It overrides the {Project} variable and becomes the current project for this command.
-m	All "mounted" root projects. Display or set the checkout directories for all root projects currently mounted.
-r	Recursively display or set checkout directories.
-x	Reset the checkout directory back to the default, i.e. the current directory ":".

Examples The following command causes subsequent files in the current project to be checked out to the HD:work:sort directory.

```
CheckOutDir HD:work:sort
```

The next command outputs the checkout directory of the current project in the form of a CheckOutDir command.

```
CheckOutDir
CheckOutDir -project MPW∫Tools∫Sort  HD:work:sort
```

To -r option allows the user to display the checkout directory for the current project and all subprojects. In this case only the sort project has a checkout directory setting that differs from the default.

```
CheckOutDir -project MPW∫ -r
CheckOutDir -project MPW∫ :
CheckOutDir -project MPW∫Shell :
CheckOutDir -project MPW∫Tools :
CheckOutDir -project MPW∫Tools∫Sort HD:work:Sort
```

The -r option can also be used to set the checkout directories of a complex project to mirror the projects own hierarchical structure. For example:

```
CheckOutDir -p MPW∫ -r HD:Work:
```

After executing the above command, listing the checkout directories for the projects under MPW would yield:

```
CheckOutDir -project MPW∫ -r
CheckOutDir -project MPW∫ HD:work:
CheckOutDir -project MPW∫Shell HD:Work:Shell
CheckOutDir -project MPW∫Tools HD:Work:Tools
CheckOutDir -project MPW∫Tools∫Sort HD:Work:Tools:Sort
```

Notice how the directory structure is similar to the project structure.

The "-m" option lists the checkout directories of the root projects. For example:

```
CheckOutDir -m
CheckOutDir -project MPW∫  HD:Work:MPW
CheckOutDir -project Test∫  HD:Test
```

See Also MountProject

DeleteNames - delete user-defined symbolic Names

Syntax

DeleteNames [-u *user*] [-project *project*] [-public] [-r] [*names*... | -a]

Description

Delete the specified Names from the current project. Names are spilt into two categories, public or private. Either kind can be deleted with this command. Special care should be used when deleting public names because once deleted they cannot be recovered.

Input

None.

Output

None.

Diagnostics

Errors and warnings are written to diagnostic output.

Status

The following status values are returned:

0	No Errors
1	Syntax Error
2	Error in Processing
3	System Error

Options

-u *user*	Name of the current user. This overrides the {User} shell variable.
-project *project*	The name will be deleted from this project.
-public	The specified names are public names
-a	Delete all private names, or public names if "-public" option is present.
-r	Delete names recursively starting with the current project.

Examples

The following example deletes the name "workingSet" from the private name space.

```
DeleteNames -p MPW∫Tools∫Sort workingSet
```

This example deletes the same name recursively starting with the Tools project.

```
DeleteNames -p MPW∫Tools -r workingSet
```

See Also

NameRevisions

DeleteRevisions - delete revisions and branches

Syntax

DeleteRevisions [-u *user*] [-project *project*] [-checkout] *revision* ...

Description

For each specified revision delete all the previous revisions on that branch. If a branch name is specified then the entire branch will be deleted. It is an error to try to delete a revision that is currently checked out for modification. It is also an error to attempt to delete a branch that has one or more revisions checked out for modification.

Revision is either a Name, the name of a file in the current project, or a filename followed by a comma and a revision number.

Checkouts (for modification) can also be canceled using this command and the "-checkout" option. If a revision has been checked out for modification and for some reason the file cannot be checked back in (the file was lost) then use this option to cancel the checkout.

Warning! DeleteRevisions permanently removes the revisions and branches specified. They cannot be recovered.

Input

None.

Output

None.

Diagnostics

Errors and warnings are written to diagnostic output.

Status

The following status values are returned:

0	No Errors
1	Syntax Error
2	Error in Processing
3	System Error

Options

-u *user*	Name of the current user. This overrides the {User} shell variable.
-project *project*	Name of the project that contains the files. This overrides the {Project} variable and becomes the current project for this command.
-checkout	Cancel the checkout on the specified revisions.

Examples

The following example deletes all the revisions before the latest in file.c in the named project.

```
DeleteRevisions -project MPW∫Tools∫Sort file.c
```

The following example deletes all the revisions on branch 22a in file.c of the current project.

```
DeleteRevisions file.c,22a
```

Suppose that revision 5 of interface.c (in project Sort) is checked out for modification and somehow the file is lost. The project expects the file to be checked back in at some point creating revision 6. But since the file is gone that will never happen. Work sould continue by making a branch off revision 5 and working on the branch, but at some point you'd like to get back on the main trunk. This can be done by the following command:

```
DeleteRevisions -checkout interface.c,5
```

Now work can procedd on the main trunk as if revision 5 was never checked out.

ModifyReadOnly - allow modifications to a read-only file

Syntax ModifyReadOnly *file*

Description The ModifyReadOnly command allows read-only Projector files to be subsequently modified. After executing this command on the desired file any modifications can be made. The file can then be checked back into the project.

This command is very useful when the project cannot be accessed in order to check a file out for modification. One of the dangers is that several people can then make changes to the same revision forcing someone to merge together all the changes.

Input None.

Output None.

Diagnostics Errors and warnings are written to diagnostic output.

Status The following status values are returned:

0	No Errors
1	Syntax Error
2	Error in Processing
3	System Error

Examples The following command allows the user to edit main.c

```
ModifyReadOnly FS:MPW:main.c
```

The next command makes the active window editable

```
ModifyReadOnly "{Active}"
```

After making the modifications to the active window it can be checked in as follows

```
CheckIn "{Active}"
```

When this file is checked in it trys to become the next revision on its branch. If the file was revision 5 then the checkin attempts to create revision 6. However, someone else may have already created revision 6 since revision 5 was not locked by the user. In this case the above checkin will fail. A second alternative is to check the file in on a branch as follows

CheckIn -b "{Active}"

This will create a new branch off revision 5. Another possiblity is to check out revision 6 for modification and merge the changes into revision 6 and check in this new file creating revision 7.

MountProject - mount an existing project

Syntax

MountProject [-d] [*project*]

Description

MountProject adds *project* to the root project list. *Project* is the HFS path of the project directory for the project. Once a project is added to the root project list, it and all of its subprojects can be accessed.

MountProject commands typically appear in the UserStartup file in order to automatically initialize the root project list.

If *project* is omitted, then the root project list is written to standard output in the form of MountProject commands.

To remove a project from the root project list, use the "-d" option. To permanently remove the project from the list delete the corresponding MountProject command from the UserStartup file.

Note that this command has no -u (user) option. Since the root project list is part of the MPW environment, which is not partitioned on a per user basis, a user name is not required.

Input

None.

Output

If no parameters are given, MountProject outputs the list of root projects.

Diagnostics

Errors and warnings are written to diagnostic output.

Status

The following status values are returned:

0	No Errors
1	Syntax Error
2	Error in Processing
3	System Error

Options

-d	Remove the named project from the project list. The specified project must be a root project. **NOTE:** The name should be a project path, not the corresponding HFS path. The next release will support either.

Examples

The following MountProject commands add the projects MPW and sort to the root project list.

```
MountProject  FS:MPW
MountProject  HD:localProjects:sort
```

To obtain a list of the current root projects, execute the MountProject command without parameters.

```
MountProject
MountProject   FS:MPW
MountProject   HD:localProjects:sort
```

To remove the MPW project from the project list use the "-d" option.

```
MountProject -d MPW
```

The following will remove all projects from the project list.

```
MountProject -d
```

Suppose under the MPW project there is a subproject called Tools. Mounting the MPW project gives access to all the subprojects simply by specifying the full project path of the project. Mounting a project that is already a subproject of a mounted project brings that project to the level of a mounted project. For example:

```
MountProject FS:MPW:
```

The Tools project is named MPW∫Tools. But after executing:

```
MountProject FS:MPW:Tools
```

the Tools project can be accessed by the name Tools.

NameRevisions - name files and revisions

Syntax NameRevisons [-u *user*] [-project *project*] [-public | -b] [-e] [-r] [-s] [*name* [*names*... | -a]]

Description Create *name* to represent a set of revisions. Subsequently, when *name* is used in Projector commands, its value, *names,* will be substituted in its place. Names are kept on a per project basis and can be composed of file names, revisions, branches and other defined Names. A Name can only include one revision per file. The first character of a Name cannot be a digit (0-9). Also, commas, greater than or less than symbols ('<', '>'), or dashes ('-') are not allowed anywhere in a Name. Names are not case senstive.

The Names are partioned into two groups, public and private names. The default is to create a private Name. Include the "-public" option to make the Name available to all users. Definitions for private Names can be added to UserStartup. Public Names are stored with the project so they only need to be defined once. Do not put public Name definitions in UserStartup.

If *names* is missing then the definition for *name* is listed. If *name* is missing then NameRevisions lists all of the Names in the project. In either case, the output is in the form of NameRevisions commands. The default behavior is to only list private names.

Projector checks for various errors both when a Name is defined and when it is used. Errors include refering to a non-existent file or refering to more than one revision in a file.

Input None.

Output When *name* or *names* are missing, the command writes Names and their values to standard output.

Diagnostics Errors and warnings are written to diagnostic output.

Status The following status values are returned:

0	No Errors
1	Syntax Error
2	Error in Processing
3	System Error

Options

-u *user*	Name of the current user. This overrides the {User} shell variable.
-project *project*	Name of the project in which to create this name. This overrides the {Project} variable and becomes the current project for this command.

-public	Create a public Name. This lets all users in the project have access to the name. Without this option a private Name is defined.
-b	Print both public and private names. The option is only valid when listing name definitions as opposed to defining a Name.
-a	All the files in the project. The Name will expand to all the files in the project.
-e	Evaluate and expand Names and files to the revision level before defining the Name or listing values if *name* or *names* is missing.
-r	Recursively execute the NameRevisions command on the current project and all of its subprojects.
-s	Print a single name per line.

Examples The first example defines a Name "Work" that gets expanded to the files file.c and interactive.c.

```
NameRevisions Work file.c interactive.c
```

The following command:

```
CheckOut Work
```

Is equivalent to:

```
CheckOut file.c interactive.c
```

By omitting the Names parameter, the next NameRevisions command will output the current definition of Work.

```
NameRevisions -s Work
NameRevisions Work -u 'user name' -project 'name' ∂
 file.c ∂
 interactive.c
```

The next command creates the Name "file.c" that expands to the second revision off the first branch off the 1.1 revision of file.c.

```
NameRevisions file.c file.c,1.1a2
```

The following:

```
CheckOut file.c
```

will now check out revision 1.1a2 of file.c.

The next example creates a Name "file.c" that expands to the the first branch off the 1.1 revision of file.c.

```
NameRevisions file.c file.c,1.1a
```

So the checkout command:

```
CheckOut file.c
```

will check out the latest revision on the first branch off revision 1.1 of file.c.

The "-e" is an important option. The following two command illustrate its function:

```
NameRevisions fred file.c
NameRevisions -e fred file.c
```

The first command defines a Name "fred" that always expands to the latest revision of file.c. The second example expands to the latest revision at the time of definition. If the latest revision of file.c is revision 9, the second NameRevisions command is equivalent to:

```
NameRevisions fred file.c,9
```

No matter what new revisions are added to file.c "fred" will always expand to revision 9 of file.c.

This next example will define all the latest revisions in the project Count to be part of "v1.0 B1". By making this a global name, all users accessing the Count project will be able to use the name "v1.0 B1".

```
NameRevisions -public "vB1 1.0" -p Count -e -a
```

The name "BetaRelease" is defined recursively for all projects within the MPW project:

```
NameRevisions -project MPW∫ -r -e "BetaRelease" -a
```

Its behavior is the same as if the user executed the following commands individually:

```
NameRevisions -project MPW -e "BetaRelease" -a
NameRevisions -project MPW∫Shell -e "BetaRelease" -a
NameRevisions -project MPW∫Tools -e "BetaRelease" -a
NameRevisions -project MPW∫Tools∫Sort -e "BetaRelease" -a
...
```

See Also ProjectInfo

NewProject – create a project

Syntax

NewProject -w
NewProject -close
NewProject [-u *user*] [-cs *comment* | -cf *file*] *project*

Description

Create a project under control of Projector. This project becomes the current project. A project directory is created where the project database is maintained. All files, comments, and other information related to the project is stored within this database. The name of the directory is the name of the project.

If *project* is a projectpath (e.g. MPW∫Tools∫Sort) then Projector creates "Sort" in the existing MPW∫Tools project. In this case MPW∫Tools must be a mounted project (see the MountProject command).
NOTE (3.0 Alpha2): This doesn't quite work correctly. The project is created but Projector doesn't realize that it should be a subproject.

If *project* is a leafname (e.g. Sort) then project directory "Sort" is created in the current directory.

Finally, if *projectname* is a partial or full HFS pathname (e.g. :Work:Sort or FS:Projects:Sort) the "Sort" project is created in the HFS location specified.

If the new project is a root project, i.e. is not part of an existing project tree, it is added to the root project list. The user should add a MountProject command to the UserStartup file in order to permanently add this project to the list.

The checkout directory is initially set to the current directory (:). To change the checkout directory, refer to the CheckOutDir command.

Use the -new option to the CheckIn command to add files to the new project.

Input

None.

Output

None.

Diagnostics

Errors and warnings are written to diagnostic output.

Status

The following status values are returned:

0	No Errors
1	Syntax Error
2	Error in Processing
3	System Error

Options

-u *user*	Name of the current user. This overrides the {User} shell variable.

-cs *comment*	A short comment about the project.
-cf *file*	The comment is contained in the file *file*.
-w	Open the "New Project" window. Bring it to the front if it is already open.
-close	Close the "New Project" window.

Examples The following command creates a project "count" in the current directory. No comment is saved with the project. One can be added later using the "Project Info" window.

```
NewProject count
```

The next example creates a count project in FS:work:count. The -cf option indicates that the comment for the new project is contained in the file "info".

```
NewProject FS:work:count -cf info
```

Finally, given that the project MPW∫Tools exists and has been mounted using the MountProject command, this command creates a "count" project in the MPW∫Tools project. In this case you don't need to add a MountProject command to UserStartup, but you may want to add a CheckOutDir command to change the checkout directory for the new project.

```
NewProject MPW∫Tools∫Count -c "MPW word count tool"
```

See Also CheckOutDir
MountProject

Project - set or write the current project

Syntax Project [-q | *project*]

Description If specified, *project* becomes the current project. Otherwise the full project name of the current project is written to standard output.

Input None.

Output If no *project* is specified, the full project name of the current project is written to standard output.

Diagnostics Errors and warnings are written to diagnostic output.

Status The following status values are returned:

0	No Errors
1	Syntax Error
2	Error in Processing
3	System Error

Options

-q	Don"t quote the project name the is written to standard output. Normally, a project name is quoted if it contains spaces or other special characters.

Examples The following Project commands set the current project to the Sort project.

```
Project  MPW∫Tools∫Sort
```

The following command will write the name of the current project to standard output.

```
Project
MPW∫Tools∫Sort
```

OrphanFile - orphan files from a project

Syntax	OrphanFile *file...*
Description	Remove any association between a file and the project from which it was checked out. *File* may be specified with a complete or partial pathname.
Input	None.
Output	None.
Diagnostics	Errors and warnings are written to diagnostic output.
Status	The following status values are returned:

0	No Errors
1	Syntax Error
2	Error in Processing
3	System Error

Options None.

Examples

```
OrphanFile HD:MPW:MyWork:file.c
HD:MPW:fileTypes.h
```

Disassociate the files: HD:MPW:MyWork:file.c and HD:MPW:fileTypes.h, from their respective projects.

```
OrphanFile ≈.[ach]
```

Disassociate all the assembly, C, and C include files in the current directory from their respective projects.

See Also TransferCkid

ProjectInfo - list project information

Syntax

ProjectInfo [-project *project*] [-comments] [-revisions] [-f] [-r] [-s] [-only | -m] [-af *author*] [-a *author*] [-df *dates*] [-d *dates*] [-cf *pattern*][-c *pattern*] [-t *pattern*] [-n *name*] [-log] [*object...*]

Description

For each project, list the the current state of all the files within that project (subprojects,by default, are ignored). For each file, list the current state of the file. For each revision, list information on that revision. If no *objects* are given, list the current state of all files within the current project.

If *object* is a projectpath (e.g. MPWƒSortƒfile.c or MPWƒSort) then Projector lists information about file.c or the files in MPWƒSort, respectively.

If *object* is a leafname (e.g. file.c), then Projector looks in the current project for the file. If the file is not a member of that project, then Projector looks for the file in the current directory. If the file exists and is part of a project then the current state of that file is listed. Projector can determine if a file belongs to a project because that information is maintained in the resource fork of all checked out files.

Finally, if *object* is a valid partial or full HFS pathname of a file, and the file is part of a project, then the current state of that file is listed.

To list the contents of a specific revision of a file, append a comma followed by the revision number to the filename specified, e.g. revision 22 of file.c is specified as file.c,22.

The **-af**, **-a**, **-df**, **-d**, **-n**, **-cf**, **-c**, **-t** options may be used to constrain the information listed to specific authors, dates, names, or containing specific comments or tasks.

Input

None.

Output

Information is written to standard output.

The following template shows the information listed in ProjectInfo.

```
Project Name
        filename,revision
       Author: author of current revision
       Status: Date
       Task:
       Comment:
```

The first line lists the project name to which the file or revision belongs. The project name is listed only at the beginning of the file or revision list corresponding to that project. The filename is something like file.c. The revision is the latest revision of filename by default. If the **-revisions** option is used, then all revisions will be listed. A "+" on the revision indicates that it is currently checked out. The status will be either "Checked in" or "Checked out". The date is the date and time corresponding to the checkin or checkout of that revision. The task lists the task associated with that file or revision, and the comment is an optional field included with the **-comments** option.

Status

0	No Errors
1	Syntax Error
2	Error in Processing
3	System Error

Options

-u *user*	Name of the current user. This overrides the {User} shell variable.
-project *project*	Name of the project that contains the files. This overrides the {Project} variable and becomes the current project for this command.
-revisions	List the revision history for each file specified.
-r	Recursively list all subprojects encountered, i.e. list every file in every subproject.
-s	Short listing.
-f	List file names.
-comments	Include comments associated with each project, file and revision listed. Normally, they will be ommitted.
-m	Only list modifiable files or revisions.
-log	Print the log information for the current project. The log contains information about the creation and deletion of public names, and the deletion of revisions.
-only	Only list information about projects and subprojects in the current or named project, i.e. not files.
-af *author*	Only list those files whose author is *author*.
-a *author*	Only list those revisions whose author is *author*.
-df *dates*	Only list those files which were created during *dates*. *Dates* can take the following forms:

Format	Meaning
date	On *date*
<date	Before but not including *date*
≤date	Before and including *date*.
>date	After but not including *date*.
≥date	After and including *date*.
date1-date2	Between and including *date1* and *date2*

A date is specifed as mm/dd/yy [[hh:mm[:ss] AM|PM].

Note: Be sure and quote *dates* so that the MPW Shell does not interprete any of the special characters.

-d *dates*	Only list those revisions which were created during *dates*.
-cf *pattern*	Only list those files whose comments contain *string*. A string may be a literal string, or a regular expression enclosed in slashes (/).
-c *pattern*	Only list those revisions whose comments contain *string*.
-n *name*	Only list those files whose revisions have the Name *name*. *Names* can take the following forms:

Format	Meaning
name	With Name *name*
<*name*	Before but not including *name*
≤*name*	Before and including *name*.
>*name*	After but not including *name*.
≥*name*	After and including *name*.

Note: If any of the name relations are used (<, ≤, >, ≥) quote *name* so that the MPW Shell does not interprete the special characters.

-t *pattern*	Only list those revisions whose task fields contain *string*.

Examples

In the example below, the current project has three files. The presence of the plus (+) indicates that Bob currently has revision 22 of file.c checked out for modification, and Peter has revision 33 of hdr.c checked out for modification. The date field of these two files reflects the date-time they were checked out. Since no plus appears on the line for file.h it can be checked out for modification. Its latest revision is 17 and the author of the revision was Bob.

```
ProjectInfo
Sample∫
file.c,22+
     Owner: Bob
     Checked out: Fri, Apr 8, 1988, 3:45 PM
     Task:  Fixing bug #223
file.h,17
     Author: Bob
     Checked in: Mon, Apr 4, 1988, 10:10 AM
     Task:
hdr.c,33+
     Owner: Peter
     Checked out: Tue, Apr 12, 1988, 5:58 PM
     Task:  Fixing bug #333
```

Using the -only option causes ProjectInfo to only list information about the project itself.

```
ProjectInfo -only
Sample∫
     Author: Bob
     Create date:  Mon, Apr 4, 1988 8:20 AM
     Mod date: Thu, Apr 14, 1988, 6:00 PM
```

Using the -f option causes ProjectInfo to list file names. Note that revision numbers are absent and the file's author and last-mod-date are listed. In the example below, file.c and hdr.c are currently checked out.

```
ProjectInfo -f
Sample∫
file.c
     Author: Bob
     Create date: Mon, Apr 4, 1988, 10:00 AM
     Mod date: Tue, Apr 5, 1988, 2:15 PM
     Free: No
file.h
     Author: Bob
     Create date: Mon, Apr 4, 1988, 10:00 AM
     Mod date: Mon, Apr 4, 1988, 10:00 AM
     Free: Yes
hdr.c
     Author: Peter
     Create date: Mon, Apr 4, 1988, 3:30 PM
     Mod date: Mon, Apr 4, 1988, 6:00 PM
     Free: No
```

Using the -f and -s options together cause ProjectInfo output the list of files in the project.

```
ProjectInfo -f -s
Sample∫
file.c
file.h
hdr.c
```

Using the -revisions option when naming a project or a file causes the revision history of the file to be displayed. Note that the comment option has been included here as well.

```
ProjectInfo -revisions -comments file.c
file.c
     Revision 22+
            Owner: Bob
            Checked out: Fri, Apr 8, 1988, 3:45 PM
            Task: Fixing bug #223
            Comment: COMMENT...
     Revision 22
            Author: Bob
            Checked in: Thu, Apr 7, 1988, 1:10 PM
            Task: Fixing bug #222
            Comment: COMMENT...
```

```
    Revision 21
          Author: Bob
          Checked in: Mon, Apr 4, 1988, 9:25 PM
          Task: Updating procedure comments
          Comment: COMMENT...
...
```

Information about HFS files may be displayed by specifying a partial or full HFS pathname. This displays the information in the 'ckid' resource of the file.

```
ProjectInfo :file.c
:file.c,22+
    Owner: Bob
    Project: Sample∫
    Checked out: Fri, Apr 8, 1988, 3:45 PM
    Task: Fixing bug #223
```

In the example below, only revisions created by "Bob" and created on or after April 4, 1988 are displayed.

```
ProjectInfo -revisions -a Bob -d "≥4/4/88"
Sample∫
file.c
    Revision 22+
          Owner: Bob
          Checked out: Fri, Apr 8, 1988, 3:45 PM
          Task: Fixing bug #223
    Revision 22
          Author: Bob
          Checked in: Thu, Apr 7, 1988, 1:10 PM
          Task: Fixing bug #222
    Revision 21
          Author: Bob
          Checked in: Mon, Apr 4, 1988, 9:25 PM
          Task: Updating procedure comments
file.h
    Revision 17
          Author: Bob
          Checked in: Mon, Apr 4, 1988, 10:10 AM
          Task:
```

In the example below, only revisions that have a task dealing with "Bug #222" are listed.

```
ProjectInfo -revisions -t /bug≈222/
Sample∫
file.c
    Revision 22
          Author: Bob
          Checked in: Thu, Apr 7, 1988, 1:10 PM
          Task: Fixing bug #222
hdr.c
    Revision 31
          Author: Peter
          Checked in: Fri, Apr 1, 1988, 3:50 PM
          Task: Bug222 - Adding check procedure
```

The final example shows the log option.

```
ProjectInfo -log
TheShell∫Projector
     Author: Peter J. Potrebic
     Create date: Mon, Apr 4, 1988, 1:59 AM
     Mod date: Wed, Jul 6, 1988, 10:35 AM
  7/5/88 4:07 PM
     Peter J. Potrebic
     DeleteNames Work
  7/2/88 1:37 PM
     Peter J. Potrebic
     NameRevisions Work bitmaps.a,2 ckid.c,3a2
```

The log shows that Peter created a public name on July 2 and then deleted it on July 5.

See Also MountProject

TransferCkid - move a ckid from one file to another

Syntax TransferCkid *sourceFile destFile*

Description Move the Projector ckid associated with *sourceFile* into *destFile,* and then remove the ckid from *sourceFile. SourceFile and destFile* may be specified with a complete or partial pathnames.

Input None.

Output None.

Diagnostics Errors and warnings are written to diagnostic output.

Status The following status values are returned:

0	No Errors
1	Syntax Error
2	Error in Processing
3	System Error

Options None.

Examples

```
TransferCkid oldFile.c newFile.c
```

Move the ckid from old`File.c` to newFile.c. Once the transfer is complete, Projector will only recognize newfile.c as belonging to a project. In addition, Projector will consider newFile.c to be the same file as oldFile.c.

See Also OrphanFile

SADE -- Symbolic Debugger Project

Alpha 2 Release Notes July 8, 1988 Fred Forsman m/s 27-E

Summary.

These notes provide information about how to use the MPW 3.0 Alpha 2 release of SADE (the Symbolic Application Debugging Environment).

This is the first SADE release to include the manual (as opposed to the ERS). These release notes, however, should definitely be read because they provide installation information and additional tips and hints in the use of the debugger based on the problems our early users have experienced. The section "How to Use SADE" below tells how to set up SADE and the section "More Information on Using SADE" gives some pointers on where to find information on using SADE without going to the manual. The command section of the manual, however, is up-to-date and can serve as a reference on matters not covered in the builtin help.

Please direct your comments on this release to one of the debugger team: Fred Forsman (x2520), Russ Daniels (x4568), John Paulson (x4163), Ira Ruben (x2002), or Burt Sloane (x6252). Feel free to give us feedback on any and all aspects of SADE (e.g., user interface, command language, command output formats, etc.).

How To Use SADE.

Installation. You may drag the SADE folder from the release disk to anywhere you like on your system (it does not need to be associated with the MPW folders). Note that the SADE application also has auxiliary files which should be kept in the same folder as SADE itself; these are SADEStartup, SADEUserStartup, SADE.Help and the SADE Worksheet. A copy of the SysErrs.Err file should be kept either in the SADE folder or else in your System Folder so that SADE can report system-related errors with textual messages. The SADEScripts folder which is included in the release is not essential to the operation of SADE and may be placed anywhere or omitted altogether.

SADE requires a special version of MultiFinder in order to access and control processes. The version of MultiFinder included with the SADE release is compatible with the latest System release (6.0). The additional code to support debugging in this version of MultiFinder should not affect your normal (non-debugging) use of MultiFinder. To install this new version of MultiFinder, drag your current MultiFinder to some place other than your System Folder, then copy this new MultiFinder into your System Folder, and then reboot.

If you do not install the special version of MultiFinder needed for SADE (or if you inadvertently replace it by updating your system) SADE will crash into Macsbug when you launch it (we will be trying to improve this graceless behavior). You can verify that your version of MultiFinder was the problem if SADE crashed on invocation by switching back to the main screen and observing whether there is a SADE window with a "TWRegisterDebugger failed!" message in it.

DO NOT run SADE on a system with QuickKeys installed. QuickKeys is known to hang the system when tracing traps in SADE.

Launching and Entering SADE. SADE should be launched like any other application, i.e., by double clicking on the application icon, by double clicking on a SADE document, or by launching it from the MPW shell.

You should be able to switch to SADE using the usual MultiFinder process switching mechanisms. If you want to enter SADE from the context of another program, hitting the NMI button should transfer you to SADE which will either display the program source or a message of the form "Program interrupted at <location>".

The target application you wish to debug with SADE may be launched from the Finder or from SADE via the "launch" command. In order to inspect the target application with SADE, the target application must be suspended. The application can be suspended by being interrupted with the NMI button, by hitting a breakpoint, or by raising an exception which SADE handles. In this release of SADE, this means that you will typically first suspend your application by hitting the NMI button, and subsequently by setting breaks. (We hope to support setting breakpoints in code segments applications which have not yet run shortly, which would eliminate the need for initially breaking with an NMI.)

SADE and MacsBug. MacsBug is still available while SADE is present. While SADE will take over the NMI button, MacsBug can be entered via any $A9FF or $ABFF trap. There is an F-Key ("DebugFKey.r") which can be easily installed with Rez into your System to generate one of these traps.

Symbolic Information. The symbolic program information support i[illegible] release of SADE is working well for C and Pascal programs. Symbolic debugging information can be generated by using the "-sym on" compiler option and the "-sym on" linker option. The ".SYM" file generated by the linker should be in the same directory as the target program; otherwise the symbol file can be identified with the "Target" command.

SADE is also able to fall back to the MacsBug symbol mechanism which identifies the names of loaded procedures and functions whose names are embedded in the code. SADE uses MacsBug information for display purposes only; that is, Macsbug names may appear in the output of the "disasm" and "stack" commands and in the string returned by the "where" function. However, SADE does not use MacsBug information on command input, so you may not use a MacsBug name (unless the name is also defined in the ".SYM" file) to set a breakpoint or to request a disassembly.

Symbolic information is available for most Mac system (ROM and low memory) symbols.

Note that Lib has been changed to support symbolic debugging information, so that object files passing through Lib will no longer lose their symbolic information.

Source Level Debugging. SADE now supports source display, allowing the current execution point (PC) to be identified in your source, and permitting the setting and unsetting of breakpoints by identifying points in the source. To allow this mechanism to work you should either keep a copy of your sources in the current SADE directory or else use SADE's "SourcePath" command to identify where your source files are located.

Source level debugging is provided through a "SourceCmds" menu which allows setting and unsetting breakpoints, single stepping by source statement, "go til", display of variable values, etc. The menu commands are also available through command keys. The "SourceCmds" menu includes a final item which allows switching between source level (statement-oriented) and assembly level (instruction-oriented) debugging. The "step" and "step into" menu items and command keys will change their behavior depending on the setting of the debugging level, as will the debugger displays when entering the debugger after program execution.

The "SourceCmds" menu includes "Break" and "Unbreak" commands which work on the current selection in the active window. There are "Step" and "Step Into" commands which work either at the source statement level or the assembly instruction level (depending on the setting of the final

"SourceCmds" menu item). The "Go" menu command begins program execution, and the "Go Til" command begins program execution with a temporary break set at the statement indicated by the current selection in the active window. The "Where PC?" item will bring up the source corresponding to the current PC, while the "Show Where" item will put up an alert indicating the procedure and statement (number) of the selection in the active window. The "Show Value" menu item allows you to select variable names in your source and then use the menu command to display the variable value; the value will be displayed in a "Values" window which is opened in the current directory.

The support for this source debugging capability is implemented in the SADEStartup file by means of several procedures in the debugger's language and some AddMenu commands; consequently, the source debugging mechanism can be modified to suit your own tastes if you spend some time investigating how it is implemented in the SADEStartup file. It is easy, for example, to disable the alert which identifies the procedure/function and statement number where execution has been suspended. To change this behavior you need to modify the "StandardEntry" proc definition which appears in the SADEStartup file. This debugger proc handles all display upon entry into the debugger; it plays this role because it is the designated action in the "onEntry" command in the startup file. (You do not need to restart SADE to have a new "StandardEntry" proc definition take effect; just select your edited definition and hit Enter.)

Another characteristic of source display that can be changed easily, is whether source windows will be brought up as the topmost window. The standard behavior in this regard has changed in this release so that source windows are not brought up as the topmost windows so that you can continue to issue commands conveniently from a worksheet window. If you want the source windows to appear on top, all you need to do is enter an assignment of the form:

```
        SourceInFront := 1
```

If you want to change this behavior for future debugging sessions you can modify the definition of "SourceInFront" in the SADEStartup file. (Let us know if you think that the default for this behavior is set incorrectly.)

More Information on Using SADE.

The following are some places to look for information on how to use SADE:

- The "Help" command works, providing on-line information about the command language.
- The SADE Worksheet file contains a variety of commands which demonstrate aspects of the command language which you may be likely to use. The Worksheet also contains a sequence of commands outlining how to debug MPW tools.
- The SADEStartup file contains a number of debugger Proc definitions and AddMenu commands which implement a source level debugging interface.
- The "SADEScripts" folder in this release contains some examples of the use of various debugger language constructs. Some of the examples provide implementations of builtin SADE commands using the SADE command language. Some information on these scripts is provided in a following section of these notes.

The following points should be noted in using SADE:

- All traps are identified in SADE by names beginning with underscores (as in Traps.a). Trap names without the underscores represent either glue or ROM addresses. Thus _InitGraf represents the trap while InitGraf represents an address in the ROM.

- SADE looks up symbol references based on the context in which program execution was stopped. In the simplest case, if we have broken inside of procedure "foo", then the name "foo" and the names of any variables declared within "foo" will be recognized in their simple or unqualified form. The names of other procedures and globals in the compilation unit containing "foo" will also be known. If you want to refer to something which is not defined within the context of the point where execution was suspended you may have to qualify the symbolic reference to allow SADE to find the symbol. For example, if you want to refer to an embedded Pascal procedure "snuk" which is inside of procedure "bar", you may specify "bar.snuk". If it is necessary to identify the compilation unit that a procedure is in then a reference of the form "\UnitOrFile.Procedure" is required. (The "\" indicates that the following name will be a Pascal unit name or C compilation file name, i.e., a "top level" name.) If you hit NMI and end up in ROM or other system code you will have to make such a qualified reference to talk about objects in your program.

- If you hit NMI and end up in ROM or other system code and want to get back to your program, you have several alternatives, some of which will take longer to execute than others. The simplest is to use the "step" command to single step until it reaches the next statement in your program -- which can take a while depending on how much and what system code SADE must step through. Alternately, you can bring up one of your source files, set a source break, and go, which should be a much faster operation. If you want to get back to the next statement to be executed in your program, you should use the "stack" command to see what the last program statement is in your call chain and set a break on the next statement and go. (We are working on ways to speed up stepping in cases such as these.)

- The first (0'th) statement of a procedure or function corresponds to the entry point to that routine before the LINK instruction has been executed, at which time the stack frame has not been set up and the variables and parameters do not have meaningful values. Consequently, if you want to check the value of a parameter you should go to statement 1 (e.g., "foo.(1)", instead of "foo" or "foo.(0)") before inquiring about the value.

- Symbolic debugging information from the Pascal compiler does not include information about the use of WITH statements, consequently, SADE does not know how to deal with unqualified field references. To access a field of a record which has been identified with a WITH statement you will need to include the name of the record variable (as you would if the WITH were not present).

- In SADE a variable reference always refers to its value, and not its address; thus, "foo" refers to the value of the variable "foo" and "@foo" refers to its address. The "Dump" command takes an expression argument to specify what address to dump from, thus, if you want to dump the value of "foo" you should specify "dump @foo" (since "dump foo" will interpret the value of foo as an address and display the memory at the address indicated by foo's value).

- The values of expressions in SADE are long values by default. If some other size is desired a type coercion should be used. Thus the following "find" command will look for a long value
  ```
      FIND $ABCD PC 20
  ```
  while
  ```
      FIND word($ABCD) PC 20
  ```
  will look for a word-sized value.

- The names of procedures and functions (and statement references relative to procedures) can be specified as arguments to the break and other commands, as in "break foo";

however, if you wish to assign the address of a procedure to a register you should precede the procedure name with the @ operator, as in "pc := @foo". In the former case the procedure name (or statement reference) represents a special code reference (with extra information about resources and offsets), while in the latter it represents a simple address (at a specific point in time).

Known Problems In This Release.

The following features described in the ERS are not yet implemented:

- PMMU registers are not supported (and perhaps never will be), and 68030 transparent translation registers are also not supported.
- Setting breakpoints in unloaded code segments is now supported; however, you cannot issue SADE commands which refer to programs which are not yet running.
- The "Call" command is not yet implemented. It is likely that "Call" will not be implemented for the first release
- While structured program variables may be assigned to SADE variables, and the SADE variables may be subsequently displayed as structured types, field dereferences may not be applied to SADE variables with structured values.
- The two machine remote-debugging configuration is not supported (and won't be in this release of SADE).
- Display of Pascal sets is not supported, but is handled better than in the Alpha 1 release. Display of records containing sets will not terminate when the set is reached.

You may notice problems with the following:

- The definition of source statements is still evolving. E.g., a breakpoint set on a WHILE will only be encountered once on entry (setup) into the WHILE. This and related problems are being addressed by further refinement of the compilers' notion of statements. You will also notice that SADE sometimes will select several statements as if they were a single statement. This results from statements for which there was no generated code, resulting in several statements which map to the same address. The statements grouped together in such cases are not always intuitive, so steps are being taken to improve the display.
- The display of your application's windows may look strange after transferring control to SADE and back. Do not adjust your set; this is not actually a problem. Remember that your application gets suspended and does not receive any update events for windows which are obscured while you are in SADE. It is possible that this problem can be improved in the future if we are able to get additional support from MultiFinder to save the window bits for suspended applications.
- The "system symbol" information (i.e., ROM and low memory names) which are built into the Alpha version of SADE are for the Mac II. By Beta we will provide support for SE and Mac Plus systems.
- Variables which are declared as EXTERN in one file but whose defining file was not compiled with symbol information will not be known to SADE. For a variety of reasons (including the desire to reduce the proliferation of duplicate definitions in and resulting

expansion of symbol information) EXTERN definitions do not produce symbolic information for the debugger.

This Alpha 2 release has the following known bugs:

- The "Kill" command functions correctly by itself, but will terminate any other pending SADE commands if issued from within a structured construct such as a begin...end. If you kill a target with breakpoints still in place and then relaunch it, SADE will report false breaks in the new invocation of the application.
- Output redirection will be cancelled if SADE is reentered a second time after first reentering via a "stop" command in a break action.
- "Unbreak all" on rare occasions leaves a functioning breakpoint around (visible with the "List break" command). A second "unbreak all" command will remove the breakpoint.

We are working on the problems described above and expect to have versions of SADE shortly which will remove some of the above limitations and problems. If you are interested in getting a more current version of SADE or if you encounter a problem you can't work around using this release of SADE, please contact one of the SADE team members.

SADE Scripts Included With This Release.

A number of SADE scripts defining procs and funcs (procedures and functions in the debugger's language) have been provided in this release. These procedures and functions can be loaded using the "execute" command on the various script files. The new commands that will then be defined provide a number of higher-level debugging functions which display information about system and program structures. (Some of these were trial implementations of what are now builtin commands in SADE.)

The following files are included:

- **DisplayMemory** has the definition of a "DM" command to display sections of memory in hex and ASCII. This facility is now built into SADE as the "Dump" command.
- **FCBChecker** has the definition of a "DisplayFCBs" command which provides a symbolic display of all of the file control blocks in the system.
- **HeapProcs** has the definitions for "HeapDisplay", "ShowFreeMP", and "DisplayHeapInfo" commands which display the heap, show free master pointers, and display summary information about the heap respectively.
- **MiscFunctions** has the definitions for "Max" and "Min" which show how to create functions with an arbitrary number of arguments.
- **MiscProcs** has the definitions for "DisplayRegs" and "DisplayWindowList" which display the machine register state and the system window list.
- **ResMap** has the definition of a "ResMap" command to display a resource map. The builtin "Resource" command performs a similar function.
- **ResVerify** has the definition of a "ResVerify" command to validate resource maps.

- **StackCrawl** has the definition of a "StackCrawl" command to display the stack of the program being debugged. This facility is now built into SADE as the "Stack" command.

Thanks to Jim Friedlander, Julia Menapace, Russ Daniels, and Fred Forsman for the above scripts. If you develop any useful and/or fascinating debugger procs and funcs of your own, please send a copy to one of the SADE team members so that it can be used as a manual example or as part of a library of useful debugging functions.

Macintosh® SADE Reference Manual

ALPHA DRAFT #2
(corrected version)

Writer: Catherine Lipson
Contact: x2755, LIPSON1
Date: July 8, 1988

NOTE: This draft contains preliminary material only. It does not contain:

- final technical or editorial changes
- final art
- final program examples
- an index

APPLE CONFIDENTIAL

Draft History:

Working Draft 1: alpha draft	April 15, 1988
Working Draft 2: alpha draft #2	June 17, 1988
Working Draft 2: corrected version	July 8, 1988

Simultaneously published in the United States and Canada.

ISBN 0-201-xxxxx-x
ABCDEFGHIJ-DO-898
First printing, Nnnn 1988

Contents

List of Figure and Tables ***for next draft***

Preface
About This Book

The SADE Reference manual describes the functionality, user interface, and command language of the Symbolic Application Debugging Environment (SADE™). This symbolic debugger allows the developer to monitor the execution of a program at the symbolic program source level and the processor level. SADE is intended primarily to debug applications and tools created with the Macintosh Programmer's Workshop (MPW). It is possible for other development systems to use SADE if they supply the required symbol information.

Hardware and software compatibility

You can use SADE to debug application programs on the Apple® Macintosh II™ computer, the Macintosh SE computer, and the Macintosh Plus computer. The size of SADE prohibits its use on the 128K and 512K versions of the Macintosh. SADE supports the Motorola MC68000 family of processors and coprocessors.

Generating program debug information during compiles and links using the MPW development system (version 3.0) requires a minimum of 1 megabyte of memory.

SADE requires MultiFinder to be installed and active. The version of MultiFinder included with the SADE release is compatible with System File 6.0. SADE supports only the Macintosh Toolbox environment; there is no support for UNIX® or other operating systems. Later versions of MultiFinder will include the needed support for SADE.

About this manual

The material in the SADE Reference Manual is aimed at readers who have a thorough understanding of the compilers, linkers, and other programming tools they are currently using. The "how to" information on writing, compiling, or linking your program is covered in the manuals specific to the tools you are using.

The primary audience for this book consists of Macintosh application developers and other professional programmers. The primary focus is on those users who are already familiar with the MPW environment. You should already be using a debugger or emulator to help you in your program development work. The chapters in Part I describe SADE features, and serve as a reference to commands and techniques you need to use often. The chapters of Part I are as follows:

Chapter 1, "SADE Overview," provides general information on the SADE interface and its many powerful features. The chapter presents the high-level interface first, with a description of the SADE Worksheet and menus. This chapter also contains a brief preview of the command-line interface, which is described in more detail in Chapter 3.

Chapter 2, "Getting Started," provides step-by-step instructions for installing the SADE files on your system. The chapter then describes the various methods used to enter SADE. The purpose of the SADEStartup file is defined here, and some simple SADE operations, such as displaying help information, are demonstrated. This chapter also describes some of the commands that you can use to control the high-level interface that SADE uses in dealing with files, windows, and menus.

Chapter 3, "Debugger Symbols and Command Language Format," provides a reference for all types of debugger symbols and expressions. This chapter focuses on the command-line interface, and the rules and guidelines you must follow to enable SADE to properly interpret symbol names. It also describes the proper format for SADE command usage, and how the SADE command interpreter evaluates commands.

Chapter 4, "Basic Debugger Operations," includes examples from the program "Eventlog" to illustrate particular commands. The chapter focuses on the commands that allow you to locate, display, or alter selected places in in your program's memory. Heap and resource commands for displaying and validating these system data stuctures are also included in this chapter.

Chapter 5, "Program Control," describes how to use SADE commands to control a target program's execution. These commands let you set breakpoints on one or more addresses, on trap ranges, or on all traps. You can also step through a program, trace program execution, or suspend execution.

Chapter 6, "Debugger Command Flow Control," contains information on the conditional, looping, and grouping constructs used within the SADE command language. These constructs are useful when you wish to automate your debugging session, because they allow you to conditionally execute or repeat a series of commands.

Chapter 7, "Customizing the Debugging Environment," tells how to write your own SADE procedures, functions, and macros. In addition, this chapter explains how you can use the SADEStartup file, the AddMenu command, and other SADE features to create a debugging environment that suits your needs. This chapter also contains information on executing debugger command files.

Chapter 8, "Source Level Debugging," describes the SADE source file interface. This source level debugging functionality is implemented through use of SADE procs within the SADEStartup file, which can be customized by the user to suit particular debugging needs.

Appendix A, "SADE Menus," illustrates the standard SADE menus, which are similar to standard Macintosh menu types.

Appendix B, "Sample Program," lists the entire sample program used as an example in this book. Appendix C, "Editing in SADE," provides an overview of the SADE editing functions.

Appendix D, "Symbol File Format ," contains a guide to the contents of a SADE symbol file, which is produced by the Linker. Appendix E, "Object File Format," provides information on the symbol output produced by the MPW compilers.

As you become familiar with the use of SADE, the introductory chapters will become less important. The command summary in Part II is designed to be a useful quick reference for experienced users. Part II contains command pages, arranged in alphabetical order.

Syntax notation

The following syntax notation is used to describe SADE commands:

terminal	Plain text indicates a word that must appear in the command exactly as shown. Special symbols (-, §, &, and so on) must also be entered exactly as shown.
nonterminal	Items in italics can be replaced by anything that matches their definition.
[optional]	Square brackets mean that the enclosed elements are optional.
repeated...	An ellipsis (...), when it appears *in the text of this reference only,* indicates that the preceding item can be repeated one or more times.
a | b	A vertical bar indicates that you can choose between the items on either side of the bar.
(grouping)	Parentheses indicate grouping. Parentheses are useful with the vertical-bar notation (|) and the ellipsis notation (...).

Filenames and command names are not sensitive to case. By convention, they are shown with initial capital letters. Terms printed in **boldface** are defined in the text and appear in the glossary.

Bibliography

Several of the chapters in this book assume you are familiar with the concepts explained in Volumes I–IV of *Inside Macintosh* (published by Addison-Wesley, 1985). For instance, you need to know how the heap and resources work on the Macintosh for the SADE heap and resource commands to make sense. Additional features of the Macintosh SE and Macintosh II computers are documented in *Inside Macintosh* Volume V (Addison-Wesley, 1988). Finally, you'll need the appropriate documentation for the programming environment and languages you'll be using.

Part I - SADE Reference Manual

Chapter 1
SADE Overview

This chapter introduces the Standard Application Debugging Environment (SADE), a program designed for high-level language debugging. SADE works within the overall program-development environment as a tool for detecting and correcting known program errors. You can also use SADE to test a program to verify that it is functioning as expected.

SADE provides a standard Macintosh window and menu interface, as well as a command-line interface with its own command language. Using SADE, you can monitor program execution at both the program-source level and at the processor level. SADE's intended use is to debug applications and tools created with the Macintosh Programmer's Workshop, but you can also use SADE independently of MPW.

This chapter first contains an overview of SADE features, and describes the relationship between SADE and your application program. The rest of this chapter concentrates on the SADE user interface, and gives you a preview of how you can use the SADE Worksheet, menus, and commands together to accomplish debugging tasks. These topics are all considered in greater detail in later chapters of the SADE Reference Manual.

About SADE

SADE, with its symbolic debugging capabilities, provides important advantages for Macintosh programmers. Using symbolic debugging means that symbols within the target application—procedure and function names, variables, and system objects such as Toolbox trap names—can easily be located from within SADE. A complete description of how SADE identifies symbolic information is contained in the Chapter 3, "Debugger Symbols and Command Language Format."

Here are some of the other important SADE features:

- A windowed display provides a familiar Macintosh interface for your debugging session. SADE can also display your source files in read-only windows.
- Debugging sessions are easy to start, because you can launch SADE like any other application. While your system is running under MultiFinder, you can easily switch to SADE.
- SADE commands can be saved to and loaded from disk, and debugger output can be saved for use outside of SADE.
- Source-level symbol references are fully supported, using symbolic program information created by using symbol options with the compiler and linker. SADE can display complex data types, listing all the fields in the structure.
- SADE includes a powerful, programmable, and extensible command language with structured control statements, user-defined procedures and functions, and full expression evaluation.
- Built-in debugger functions provide a way to perform operations such as displaying the statement line that corresponds to an address.

- Program-control facilities such as breakpoints, single stepping, or suspended execution let you see the effect of program execution on chosen program locations.
- SADE's flexible display-formatting capability allows you to choose how your debugging information will appear.
- Macintosh-specific support lets you identify heap objects and verify the consistency of heaps. Resource information and system objects, such as Toolbox trap names and low-memory global variables, can be referenced symbolically.
- SADE is easy to customize.You can define an alias for a SADE command name, add menus and menu commands, or perform other initialization routines each time you start the debugger.

These and other features are more fully described in various chapters of this manual. Many commands described in this manual are used to control execution of the target program, while other commands are used to control the workings of SADE itself. This division between program control and debugger control is reflected in subsequent chapters.

SADE and the target application

SADE is a stand-alone application, and can operate on its own within the system environment on your Macintosh. What SADE does require for any kind of meaningful use is a target application—that is, a program that you wish to debug. SADE works with the target program on two levels—on the source level, SADE can display source files, while on the processor level, SADE can display information about the program's execution.

To start a debugging session, both SADE and your target program have to be launched. You can launch first SADE, and then launch your target program, or you can launch the target program first, and then launch SADE. Once both are launched, you can return to your target program and break into the debugger.

Each time a target application is compiled and linked, symbol information can be generated in a format that SADE can interpret. The MPW C and MPW Pascal compilers generate symbolic information, controlled by qualifying options. The MPW Linker tool processes and outputs this symbol information in the form of a symbol file. See the appropriate language reference manual for information on compiler options, and the MPW Reference Manual, Chapter xx, for information on linker options.

SADE can use symbol information generated by any compiler that produces the correct object file format. The linker must also support use of symbol information. Appendix D, "Symbol File Format," and Appendix E, "Object File Format," provide a reference to the required format for compiler and linker output. Typically, the compiler and linker would include some kind of optional parameters to control symbol output. For example, a compiler might include a option for full symbol information, including source-line, variable, and type information. A compiler could also generate partial symbol information; for example, a compiler might include options to omit all source line information, all variable information, or all type information.

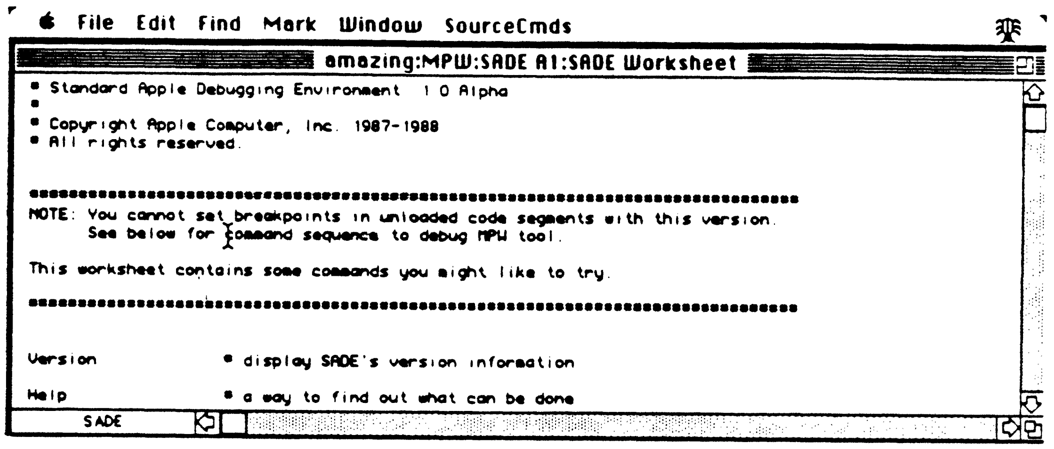
The SADE user interface

As you read in this manual, you'll soon become aware of the dual nature of the SADE user interface. Like almost every Macintosh application, SADE has windows, menus, and dialogs. When you are operating within this graphic-oriented user interface, you can do much of the work by pulling down menus, clicking the mouse, and pressing command keys. However, SADE also includes another powerful component: the command-line interface. In many respects, the SADE command language is similar to a progamming language. There are many SADE commands, each with its proper syntax. You type these commands on the keyboard, and execute them through use of the Enter key.

The sections that follow first describe the graphic-oriented user interface—the SADE Worksheet and the SADE menus. The description of the command-line interface is next, and includes a comprehensive list of SADE commands grouped by function.

The SADE Worksheet

When launched, SADE opens its own text window, known as the **SADE Worksheet.** Shown in Figure 1-1, this worksheet provides a place to enter SADE commands and display debugger output. The worksheet includes a menu bar, scroll bars, and other standard Macintosh window features. You'll see the SADE icon in the upper-right corner of the menu bar. A blinking cursor shows where you can enter commands from the keyboard. You execute the SADE commands by pressing the Enter key.



```
• Standard Apple Debugging Environment  1 0 Alpha
•
• Copyright Apple Computer, Inc. 1987-1988
• All rights reserved.


••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
NOTE: You cannot set breakpoints in unloaded code segments with this version.
      See below for command sequence to debug MPW tool.

This worksheet contains some commands you might like to try.

••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••


Version              • display SADE's version information

Help                 • a way to find out what can be done
```

Figure 1-1
The SADE Worksheet

At the window's lower-left corner, a **status panel** shows the name of the command that's currently executing, or simply the word "SADE" when you're not executing a command. A mouse click on the status panel is equivalent to pressing the Enter key.

The first time you launch SADE, the debugger opens the SADE Worksheet file. This worksheet is a text file which contains sample SADE commands and other useful information. You can enter commands and text anywhere within this worksheet.

If you decide to start debugging with a blank worksheet, you can clear the worksheet just like any other text file. For example, you can choose "Select All" from the Edit menu, and then use the Delete key to delete the contents of the current worksheet.

The SADE menus

The standard SADE menu bar includes the Apple, File, Edit, Find, Mark, and Window menus. In addition, the SourceCmds menu may appear in the menu bar, if it is generated by your SADEStartup file. (If you watch carefully when SADE starts to run, you will see that the SourceCmds menu appears just a few seconds later than the standard menu commands.)

The SADE standard menus provide an easy way to open, save, and close files, perform edit operations, and find text within files. These menus are described below.

The Apple menu includes an "About SADE" menu item, which displays the current SADE version number.

The File menu allows you to open a new or existing file, close a file, or save a file. (You can also enter several of these commands from the SADE command line: these commands include the Open, Close, and Save.) The Quit command gets you out of SADE and back to the Finder.

The Edit menu allows you to perform some of the standard edit operations on a selected portion of text within a SADE Worksheet. This manual assumes that you are already familiar with standard Macintosh editing commands, such as Cut, Paste, and Undo.

The Find menu can be used to search for a specified string. You can also perform search and replace operations from this menu.

The Mark menu allows you to set markers within a file. These markers are simply text selections identified by name, and can used to make selecting expressions easier.

The Window menu can be used to stack or tile windows. Your SADE Worksheet always appears first in the lower section of the Window menu.

The SourceCmds menu is generated by commands within the SADEStartup file. The menu includes commands to set or unset a breakpoint in the source program, clear breakpoints, step through the source program, resume program execution, display variable values, display the current program counter location, and switch between source level and assembly level debugging. You can also enter these commands from the SADE command line. You can alter or change the commands in the SADEStartup file, and add items to the SourceCmds menu if you wish. See Chapter 2 for more information on the SADEStartup file.

Many of the menu commands have Command-key equivalents, so you can speed through these standard operations once you're familiar with them. Appendix A, "SADE Menus," contains detailed information on SADE menu commands and the dialog boxes that appear when you select those commands.

You can further customize the SADE menus by placing additional AddMenu command lines in the SADEUserStartup file. Each user-defined menu command specifies a list of SADE commands that are executed when you select the menu command. See the section on "User-Defined Menus" in Chapter 7 for more information on the AddMenu command.

SADE command-line interface

The SADE command-line interface is similar to a high-level programming language. SADE commands, listed in Table 1-1, allow you to create expressions, define procedures and functions, control the flow of debugger execution, start and stop program execution, and display various symbols within your program. Many of the commands have a number of optional parameters that affect their operation. SADE also includes a number of built-in debugger functions, which you can use to perform operations such as displaying the source file that corresponds to an address. These features provide a great deal of flexibility in the command language, as well as increased power.

The rules for constructing a SADE command can be compared to the syntax requirements of a programming language. Chapter 3, "Debugger Symbols and Command Language Format," describes the components that make up SADE symbols and expressions, and explains how the SADE command interpreter evaluates a command line.

When using the command-line interface, you type one or more commands into your current SADE worksheet, and then execute the commands by pressing the Enter key. To execute more than one line of commands from the worksheet, you can highlight as many lines as you wish, and then execute the entire group of commands by pressing Enter.

You can also save a group of SADE definitions, which you use to perform a specific task, in a command file. You can then execute this file to invoke those definitions. This process is similar to using an include file with a programming language: the definitions in the file are executed once each time it is used. You can find more information on this SADE feature in Chapter 6, "Debugger Command Flow Control." The SADEScripts folder contains many examples of command files that perform useful operations.

If you wish to repeat a group of SADE commands many times during your debugging session, you can write a SADE procedure or function (hereafter known as a *proc* or *func*). You can store this proc or func within a command file. After the command file is executed for the first time, you can invoke the proc or func by name, as many times as needed. You can find more information on this subject in Chapter 7, "Customizing the Debugging Environment."

The SADE commands in Table 1-1 are grouped by function. This format reflects, to some degree, their organization within the chapters of this manual. As you read Chapters 2 through 8, you will find these commands discussed, with examples of their use. Once you reach Part II of this manual, you will find that the commands are arranged alphabetically. This change in presentation gives you two ways to find the particular command you need—by function or by name.

Table 1-1. SADE Commands

File commands		(Chapter 2)
Open	Opens a file	
Save	Saves a file	
Close	Closes a file	
Redirect	Redirects standard output	
Debugger variable commands		(Chapter 3)
Define	Defines a debugger variable	
Undefine	Removes a local debugger variable, proc, func, or macro	
Symbol command		(Chapter 3)
Symbol	Controls symbolic display in disassembly	
Application control commands		(Chapter 4)

Directory	Changes the current directory
Target	Selects program target
Sourcepath	Sets search path for source files
Launch	Launches an application
Kill	Kills an application

General expression and display formatting (Chapter 4)

Printf	Sends formatted output to file or window

Special-purpose display commands (Chapter 4)

Disasm	Disassembles instructions
Dump	Dumps a range of memory in hex and characters
Stack	Displays stack frames
List	Lists address or trap breakpoints, processes, symbols

Heap commands (Chapter 4)

Heap [display]	Displays the heap
Heap check	Verifies the consistency of the heap
Heap totals	Displays summary information for the heap

Resource commands (Chapter 4)

Resource [display]	Displays the resource map
Resource check	Checks the resource map

Searching command (Chapter 4)

Find	Searches for a target in memory

Debugger execution control commands (Chapter 5)

Shutdown	Shuts down (with restart option)
Quit	Gets out of debugger
Stop	Stops execution of debugger commands
Sbort	Stops execution of debugger commands and pending commands

Execution commands (Chapter 5)

Go	Starts execution
Step	Single steps through code

Breakpoint and tracepoint commands (Chapter 5)

Break	Sets address or trap breakpoints
Unbreak	Removes breakpoints.
Trace	Sets tracepoint
Untrace	Clears tracepoints

Flow control commands (Chapter 6)

If..end	Conditionally executes commands
While..end	Conditionally loops with beginning test
Repeat ..until	Conditionally loops with end test
For..end	Loops with a control variable
Cycle	Continues execution at top of current loop
Leave	Continues exection after the end of the current loop
Begin..end	Groups commands together

Execute debugger command file (Chapter 7)

Execute	Executes debugger commands in a file

User-defined macros, procedures, and functions (Chapter 7)

Macro	Associates characters with an identifier
Proc...end	Defines a procedure in the debugger language
Func...end	Defines a function in the debugger language
Return	Returns from a function

Menu and Alert commands (Chapter 7)

Addmenu	Associates commands with menu commands
Deletemenu	Deletes menu items
Alert	Displays an alert box

Miscellaneous commands

Help	Gets help on SADE topics or commands	(Chapter 2)
Case	Sets the case of a variable lookup	(Chapter 3)
Version	Displays current SADE version	(Appendix C)

SADE files and directories

This section describes what you will find on the SADE release disk. For installation instructions, see Chapter 2, "Getting Started."

The SADE application itself can be opened and executed just like any other application. You can recognize SADE by its distinctive "insect" icon. When SADE is running, it uses a number of other files, which are also included on the release disk. These files are as follows:

SADE.Help	This auxiliary file contains the online Help information for the SADE language. You can get this information at any time from within SADE simply by using the Help command. More information on the Help command can be found in Chapter 2.
SADE Worksheet	When you first open this file,you'll see more than an empty window. The Worksheet displays a number of typical uses for SADE commands, to give you an idea of how SADE can be used. Also included in the Worksheet is a sequence of commands for debugging MPW tools.
SADEStartup	This file contains debugger proc definitions and other commands that set up the initial SADE configuration. More information on this file can be found in Chaper 2.
SADEUserStartup	This file is initially an empy text tile. You can use the SADEUserStartup file to customize SADE startup settings.
SADEScripts	This folder contains a number of files that provide examples of debugger command usage. More information on these scripts is provided in Chapter 7.
MultiFinder	The version of MultiFinder on the SADE release disk is slightly different than the previous versions of MultiFinder. you may replace this special version with an updated version of MultiFinder sometime in the future.
SysErrs.Err	This file can be used for IO and system error messages.

Chapter 2 Getting Started

This chapter covers what you'll need to get started using SADE—how to install it, how to launch the debugger from the Finder, and how to enter the debugger from your executing application program. The SADEStartup file, which provides initialization information each time you launch SADE, is described in detail. After reading this chapter, you should be ready to start debugging with SADE.

In addition, this chapter provides a brief introduction to file-handling and command-line editing within SADE. The basic file-handling tasks include opening, closing, and saving the text files created by SADE. This chapter briefly explains how to enter and edit commands within a SADE worksheet. This part of SADE follows the standard Macintosh text-editing conventions, so you'll find much that is familiar here.

The SADE Help command is introduced in this chapter, so that as you read the remainder of this book, you'll know how to get online help. The last section in this chapter illustrates how to start up with a sample debugging session.

Installing SADE

SADE is distributed on a disk as a stand-alone application, with its own help file and other material. You can copy the SADE files into any convenient location on your hard disk. The files SADE.Help, SADE Worksheet, SADEStartup, and SADEUserStartup must be located in the same directory as the SADE application; the guidelines for placing these files are shown in Table 2-1.

Table 2-1. SADE Files

Same directory		Other directory	
SADE	Application	SysErrs.Err	I/O and system error messages
SADE.Help	Auxiliary text file	SADEScripts	Optional examples
SADE Worksheet	Auxiliary text file	MultiFinder	Place in System folder
SADEStartup	Auxiliary text file		
SADEUserStartup	Auxiliary text file		

You can create a directory for your SADE files wherever it is convenient. For example, if you want to create a folder called Debugger on your hard disk, you can simply copy the contents of the SADE distribution disk into the Debugger folder.

The SADE.Help file is a text file that is called by SADE anytime you use the Help command from your worksheet. The SADE.Help file isn't designed to be accessed directly by users.

A SADE Worksheet file is opened each time you launch SADE. Any commands you wish to enter can put anywhere within this worksheet.

You can create any number of SADE files, place commands or text in them, and give them names of your choosing. You can double-click on any SADE file to start up SADE. If you move a SADE file to another directory, and then quit SADE, you should be aware of the following behavior: when you double-click the SADE file in the other directory, SADE will launch, but the SADEStartup file won't be executed.

The commands in the SADEStartup file are executed automatically each time SADE is launched. You can change the contents of this file if you choose; see "The SADEStartup File", later in this chapter, before you attempt any changes. Any changes you make will take effect after you have quit SADE and relaunched, or after you have reexecuted the SADEStartup file.

The SADEUserStartup is an empty text file that you can use for your own commands, definitions, or procedures. This file is executed from the SADEStartup file.

You can place the SysErrs.Err and SADEScripts files in the same directory as SADE if you choose. You could also put the SysErrs.Err file in the System Folder; it's used so SADE can report system-related errors with textual messages. The SADEScripts folder contains examples for SADE command files that perform a variety of useful functions. You can move them anywhere you like, or omit them if you need extra space on your system. However, since several of the command files are used as examples in this book, it's a good idea to keep these samples around until you are well-acquainted with SADE.

The application that you are debugging can reside anywhere on the system. It's easiest to keep the symbol file generated by the linker in the same directory as the application. However, you may keep the symbol file elsewhere if you properly identify its location using the Target command.

To permit source file display, SADE must be able to find the source files for your application. If the source files aren't in the current directory, you can use the Sourcepath command to identify their location. This command is described in Chapter 4, "Basic Debugger Operations".

Remove following paragraph after final version ships

Current versions of SADE must run under a special version of MultiFinder, known as (version xxx?) To make sure that SADE will work correctly, you must first move your current version of MultiFinder out of your System Folder. Then drag the SADE version of MultiFinder into your System Folder. At this point, restart with the new version of MultiFinder; you will then be ready to try SADE.

This new version of MultiFinder is compatible with the latest System release (6.0). The code added to support debugging will not affect your normal use of MultiFinder.

Launching SADE

Once you have installed SADE, you can launch it from the Finder as you would any other application: by double-clicking on its icon, by double-clicking on any SADE document, or by choosing Open from the File menu. You can also launch SADE from the MPW shell. You can return to the Finder from your application window and launch SADE at any time.

Each time SADE is launched, it notifies MultiFinder that it is acting as the debugger for the system. MultiFinder then passes exception handling and process information to SADE. When other applications run, SADE is suspended, but as soon as an exception occurs, SADE will once again be in control. Exception handling passes back to the operating system only when you explicitly quit from SADE by using the Quit command.

When you first start SADE, the commands within the SADEStartup file execute automatically. This file contains the initialization information for your SADE debugging session. The details of the default settings for the SADEStartup file are described in this chapter. Keep in mind that these default settings may be changed if you wish to specify some other set of SADE startup actions.

Once SADE has been launched, you can begin working in a SADE window right away, or you can switch back to another application or the Finder with the usual MultiFinder process switching mechanisms. To return to SADE, just click any SADE window. The various ways to enter SADE from your application are described next.

Entering SADE from an application

This section describes several methods you can use to go from SADE to your application program, and back to SADE. The techniques described in this section assume that you have already launched the SADE application.

After SADE has been started, you can easily return to your application by clicking its window. You can then re-enter SADE from your executing application program. The most common way to do this is by means of the NMI key located on the side of your Macintosh. Also known as the programmer's switch, this key interrupts the foreground program as it's running on your system, and causes control to be passed to SADE. The SADE worksheet then comes to the foreground, and a message appears. (See the SADEStartup file for the default messages.)

If your program generates an exception due to some error, the program enters SADE when the exception occurs. Again, a SADE worksheet comes to the foreground; the worksheet may display a message indicating where the program counter was when the exception occurred. (The message displayed depends on the argument used with the OnEntry command in the startup file.)

The system error-handling routine, invoked by calling SysError, will also put you into SADE. You can use this routine within your program as a test:

```
    if (boolean expression ) SysError();
```

If the `boolean expression` becomes true while your program is executing, your program will enter SADE. This error-handling routine allows you to enter SADE from the point you choose, instead of waiting for an exception or interruption.

You can't use the debugger traps $ABFF and $A9FF for SADE, because they are reserved for MacsBug.

While you're working with SADE, you can set a breakpoint at one or more places in your program code. When you resume execution, your program will run until it reaches a breakpoint, and then control returns to SADE. The commands used to set code breakpoints are described in Chapter 5. You can also use the menu items in the SourceCmds menu to set breakpoints in a source file. See Chapter 8 for more information on source breakpoints.

The SADEStartup file

Each time you start SADE, it searches for a file named SADEStartup, and executes the commands in the file. If the SADEStartup file isn't in the same directory as SADE, it won't be found when SADE is launched. However, you can delete the SADEStartup file, or "hide" it in another directory, and SADE will still run.

You can change the commands within the SADEStartup file if you wish; however, you should be familiar with the important functions provided by this file before you attempt to modify it. This section describes the contents of the default SADEStartup file.

A debugger procedure, or proc, is the first thing you'll notice in the SADEStartup file. After this StandardEntry proc is defined, you'll see it used as the argument for the OnEntry command. This command can be used with any break action; in the SADEStartup file OnEntry provides a way for SADE to put a message in the worksheet when your program code is interrupted.

The StandardEntry proc includes a Printf statement, which shows you the cause of the interruption, the location of the Program Counter at the time the error occurred, and the name of the application program that was halted. The numeric codes listed in this debugger proc correspond to the different reasons that your program might stop executing. For example, if you press the NMI key, you will get the message "Program Interrupted". If the debugger generated an exception, as is the case in a break or trace operation, the corresponding message is displayed. These messages are described more fully in Chapter 3.

If you change the StandardEntry proc definition, you don't have to restart SADE for the new definition to take effect. You can simply select your edited definition and press Enter.

The RegisterDisplay proc causes SADE to open a window, which it places behind the active window. This window then receives the output of a number of Printf statements that display the current values of D0–D7, A0–A7, and the program counter.

The set of Macro declarations in the SADEStartup file provides a way to use MacsBug equivalent names for some SADE commands. These declarations are useful if you are accustomed to using the MacsBug command language. For more information on the Macro command, see Chapter 7.

The procs SetSourceBreak, unSetSourceBreak, and sourceStep provide a way to work back and forth between a source location and the corresponding program code in memory. These procs rely on the built in debugger functions sourceToAddr and addrToSource, which are described in Chapter 3. The AddMenu commands in the SADEStartup file cause the SourceCmds menu to appear in the menu bar for SADE. This menu makes it possible to perform these source-level debugging operations from a menu. For more information on the AddMenu command, see Chapter 7.

If you want to customize SADE, you can use the SADEUserStartup file. An Execute command in the SADEStartup file will run the SADEUserStartup file immediately after running the SADEStartup file. The Execute command is described in Chapter 7.

Entering and editing commands and text

Once you are in SADE, you'll have a SADE worksheet for typing text and commands. Each SADE worksheet is a text file, and the standard Macintosh text-editing procedures will work as you expect. The worksheet has a blinking cursor, a scrollable window, and menu names as shown in Figure 1-1.

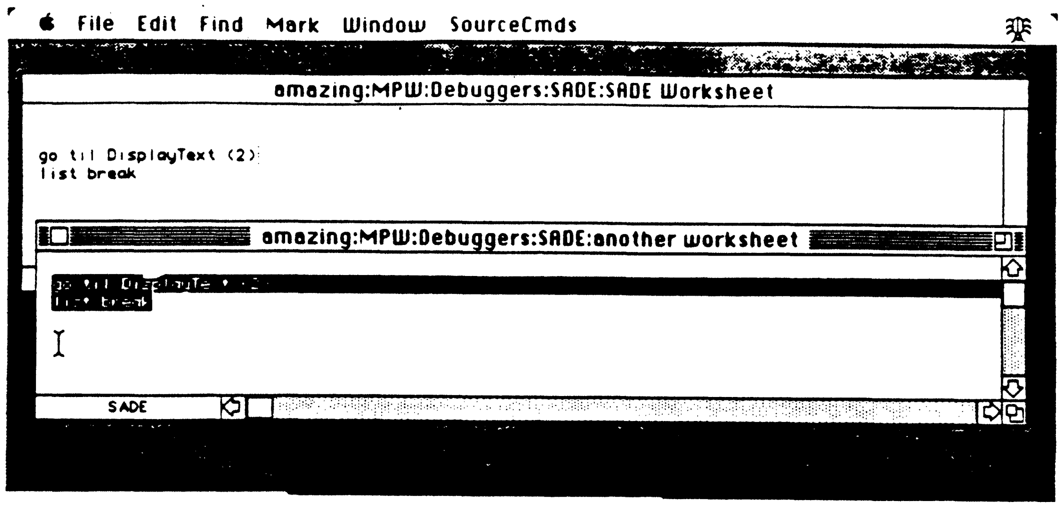
You can edit any lines of text you typed into a worksheet. When you press the Enter key, SADE executes the current line as a command line. You can enter more than one command line at a time by highlighting a series of lines, and then using the Enter key. Figure 2-1 provides an example of these two methods of command entry.

The top half of Figure 2-1 shows a worksheet with two lines of commands. The cursor is positioned at the end of the first line. If you press the Enter key at this point, SADE executes only the first line of commands.

The bottom half of Figure 2-1 shows another worksheet with the same two command lines, but this time they are highlighted. You can highlight any portion of a SADE worksheet by positioning the cursor and pressing the mouse button while you reposition the cursor at the end of the selection. Once you have highlighted a line or lines, you can press the Enter key and SADE executes all the highlighted command lines.

MPW 3.0A2 Document List

MPW Overview	• MPW 3.0A2 Release Notes
Projector	• Projector Apha 2 Release Notes • Converting Your Projects • Macintosh Programmers's Workshop Project Management System ERS "Projector"
SADE	• Alpha 2 Release Notes • Macintosh SADE Reference Manual Alpha Draft #2
Libraries	• A2 Libraries • Pre-A2 Libraries
Linker Tools	• Release Notes for MPW 3.0A2 - Linker and related tools
Interfaces	• Converting Between pre-3.0 Str255's and 3.0 Str255's • Interface Release Notes • Pre-A2 Interfaces • Macintosh Technical Notes: Setting and Restoring A5
MPW Shell	• Macintosh Programmer's Workshop 3.0 Shell ERS
Commando	• Commando's Built-In Editor and other new features
SetVersion	• SetVersion manual page
Choose	• Choose manual page • Changes to Choose
DumpFile	• DumpFile manual page
WhereIs	• Whereis manual page
Sort	• Sort manual page
Resource Tools	• Rez & DeRez ERS • Macintosh ResEdit Reference Manual Beta Draft
Parser Generator	• MPW LL(1) Parse Table Generator ERS
MacsBug	• Macintosh MacsBug Reference APDA Final Draft


File Edit Find Mark Window SourceCmds
amazing:MPW:Debuggers:SADE:SADE Worksheet
amazing:MPW:Debuggers:SADE:another worksheet
SADE

```
go til DisplayText (2)
list break
```

Figure 2-1
SADE command lines

Basic editing functions are also available as menu commands. Editing within the SADE worksheet is similar to the editing functions provided by the MPW Shell. You can select and edit text with the usual Macintosh editing techniques, using menu commands to cut, copy, and paste selected text. See Appendix C for a complete description of SADE editing functions. The Edit menu commands are further described in Appendix A.

Most of the basic file-handling functions are also available as menu commands. You can open a file by using the Open command, or by selecting its name within the worksheet and choosing the Open Selection command (Command-D) from the File menu. The file-handling commands are described in the following section,

SADE file-handling operations

When using SADE, you have a choice between using the File menu or the command line for opening, closing, and saving text files. The commands in the File menu provide the usual Macintosh interface for creating a new file, opening an existing file, closing a file, or saving a file.

The command-line interface gives you the ability to perform file operations from within a SADE command file. You can create a new text file and then redirect SADE output into it, from a command line. For instance, in the SADEStartup file, a new file named register display is created with the following command line:

```
open behind "register display"
```

The following section describes how to use the Open, Close, Save, and Redirect commands from the SADE command line. Chapter 3 presents a complete and rigorous explanation of the rules for forming SADE commands. The file-handling commands are simple enough that you will be able to use them even before you delve fully into SADE's complexities.

File commands

All of the SADE file commands work on the frontmost SADE window, unless you specify otherwise. All filenames used within a command line must be strings in quotation marks. For SADE's purposes, a window and a file are synonymous. All the windows opened by SADE function as text files (resource type 'TEXT').

When you use the Open command on a command line, its format is

open [**source**] [**behind**] *filename*

The optional **source** parameter indicates that the window to be opened is a read-only source window, as opposed to a general-purpose text window. The optional **behind** parameter causes the specified window to open behind the frontmost SADE window. Without this parameter, the window opened will become the frontmost window.

You can use the Close command to close the file you specify, or to close all files. Its format is

close [**all** | *filename*]

If you didn't save the contents of the window, SADE will ask you through a confirmation dialog whether it should save them.

The Save command will save the file you specify or, alternatively, will save all files. Its format is

save [**all** | *filename*]

The default file to be saved is the currently selected window. If you didn't modify the file since the last time you saved it, no save operation is necessary.

To redirect output from the currently selected window to another file, use the Redirect command. When you use the command on the command line, its syntax is either

redirect [**append**] *filename*

or

redirect [**pop**] [**all**]

If you used the **append** parameter, SADE appends the output from the commands entered in the currently selected window to the end of the named file. If you specify **all** or **pop all**, standard output is redirected to the SADE Worksheet. For more information on the Redirect command, see the Redirect command page in Part II.

Getting help in SADE

One of SADE's most useful features is the Help command, which provides online help. Entering Help with no parameters will display a summary of the help available, as shown here:

```
help

SADE 1.0 Help Summaries

    Help summaries are available for each of the SADE commands.
    To see the list of commands enter "Help Commands". In addition,
    brief descriptions of Variables, Constants, Expressions, built
    in functions, and Shortcuts are also included.

    To see Help summaries, Enter a command such as

    Help Builtins          # a list of builtin (predefined) variables
                           # and functions
    Help commandName       # information about commandName
    Help Commands          # a list of commands
    Help Expressions       # summary of expressions
    Help Patterns          # summary of patterns (regular expressions)
    Help Shortcuts         # summary of SADE shortcuts
    Help Variables         # summary of variable references

    Copyright Apple Computer, Inc. 1987-1988
    All rights reserved.
```

You can get help information on particular topics by using the either Help command with the name of a topic or a SADE command. For example, to get help on the List command, you can enter

```
help list

List break [traps | addrs]
List trace [traps | addrs]
List process
List symbol
```

An introduction to debugging a program

This section describes how to get SADE and your application program started. A sample program named Eventlog is used to show you how a source window looks once you have halted an executing program. Your application program may not behave in the same way as the sample program; however, the sample program will give you an idea of some possible uses for SADE.

The first step is to start SADE. For this example, assume that you have already launched SADE from the Finder by double-clicking the SADE icon.

Once you are in a SADE worksheet, you must identify the files needed for debugging, if they aren't in the current directory. The Directory, Target, and Sourcepath commands, described in Chapter 4, identify the location of your application, symbol file, and source files. The example shown in in Figure 2-2 assumes that these files are in the same directory as SADE, or that this information was already supplied.

You can launch your application from the Finder, or you can use the Launch command. In Figure 2-2, you can see the following command line in a SADE worksheet window:

```
launch ":Eventlog"
```

This command line starts the application program Eventlog. (The windows created by the running application aren't shown in this example: what you see in Figure 2-2 is a source file.) The Launch command is described in Chapter 4. Remember that you must always use quotation marks around filenames.

The next command line will cause Eventlog to stop executing on the second statement in the DisplayText procedure:

```
go til DisplayText.(2)
```

Note that a partial pathname is used in this example, identifying only the procedure name. In some cases, you'll need to use the full pathname for the program, unit, and procedure name. For this procedure, the full pathname is eventlog\TransDisplay.DisplayText.(2). See Chapter 3 for more information on pathnames for program variable references.

SADE displays an alert box that tells you that the Go Til statement worked—an address break occurred at DisplayText.(2). The Go Til command is described in Chapter 5.

The source program is displayed in the window at the top of Figure 2-2. The particular source file shown is TransDisplay.c. This file contains just one part of the Eventlog program. In the figure, the instructions on which the program stopped executing are indicated by an outlined box. When you move the cursor back into the source window, the area within the box will be highlighted.

You can now use the mouse to move the cursor between the source window and the SADE window. You can also use the SourceCmds menu to step through code, place source breakpoints, display variable values, or show the location of the program counter. These topics are all discussed in greater detail in Chapter 8.

This sample program will appear throughout the manual. Always keep in mind that it is only a sample: each of your applications will present its own unique debugging challenges.

Now that you've seen how to begin a debugging session, you're almost ready for the material in the rest of this manual. There are just a few more things you need to know about—how to quit SADE, and how to shutdown your system from SADE. These basic operations are described below.

Quitting from SADE

To exit from the SADE application, use the Quit command. This command should not be confused with the Stop and Abort commands, which operate only on the current break action. The command format is simply:

```
quit
```

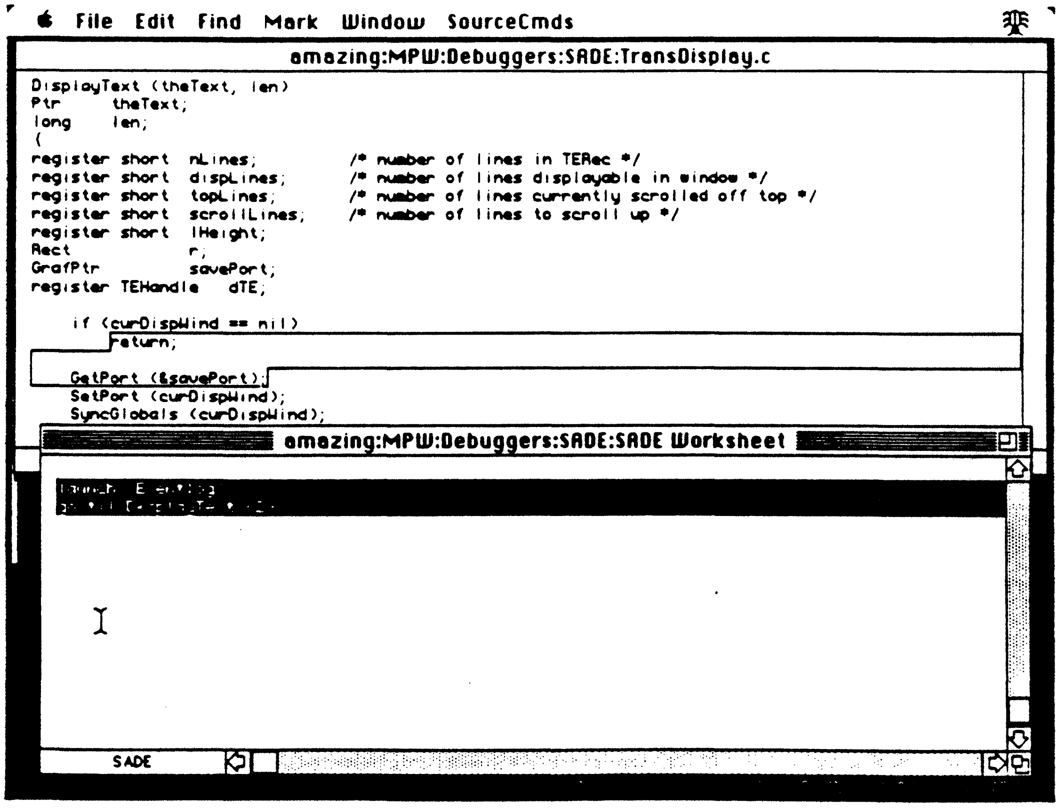
The Quit command causes the SADE itself to quit. Quit will display a dialog asking the user if it's all right to kill any suspended applications. MultiFinder is notified that SADE is no longer there as a debugger, and control is passed to another process as determined by MultiFinder.

Shutting down the system

To shut down the system from within SADE, use the Shutdown command. Its format is:

shutdown [**restart**]

The Shutdown command causes SADE to be terminated. The Shutdown Manager is called to perform the actual system shutdown. If **restart** is specified, the Macintosh will be restarted.


File Edit Find Mark Window SourceCmds
amazing:MPW:Debuggers:SADE:TransDisplay.c
amazing:MPW:Debuggers:SADE:SADE Worksheet
SADE

```
DisplayText (theText, len)
Ptr      theText;
long     len;
{
register short  nLines;          /* number of lines in TERec */
register short  dispLines;       /* number of lines displayable in window */
register short  topLines;        /* number of lines currently scrolled off top */
register short  scrollLines;     /* number of lines to scroll up */
register short  lHeight;
Rect            r;
GrafPtr         savePort;
register TEHandle   dTE;

    if (curDispWind == nil)
        return;

    GetPort (&savePort);
    SetPort (curDispWind);
    SyncGlobals (curDispWind);
```

Figure 2-2
SADE Source Display

Chapter 3

Debugger Symbols and Command Language Format

This chapter provides a reference for all types of debugger symbols and the expressions formed using them. These symbols are used within the context of the SADE command format, which is also introduced in this chapter. Although you may already have looked at some of the SADE commands, and figured out for yourself how to use a symbol or expression with a command, the background information in this chapter may answer some of your questions about how SADE interprets debugger symbols. Topics covered in this chapter include:

- How SADE determines if a symbol is a debugger, program, or system symbol, and how it detemines the proper scope for each symbol.
- The use of numeric constants, such as decimal, hexadecimal, binary, or floating-point numbers, within SADE expressions.
- The rules for using string constants within SADE expressions. These guidelines tell you what characters can be used in strings, and what the length of a string can be in a particular context.
- How to use the various types of identifiers, such as register names, predefined debugger variables, and program variables. The guidelines for working with identifiers include the distinction between a fully-qualified identifier and a partially-qualified indentifier.
- The several classes of variable references—system, debugger, and user-program variables. These variable types may be used with selector operators and escape symbols to ensure that SADE can access each type properly.
- The guidelines for using SADE expressions, including the rules for operator precedence, operand base types, expression evaluation, and type coercion.
- How SADE builtin functions such as AddrToSource and SourceToAddr can be used within expressions.

This chapter also explains how the SADE command interpreter processes SADE commands. The SADE command format sections build on the material about symbols and expressions. Other topics relating to the command language include:

- Debugger output files, which display the result of SADE commands. SADE uses one or more text files for standard output, and provides a way to redirect command output from the default output file.
- Controlling symbol display using the Symbol command. By default, all debugger output is displayed symbolically whenever possible.
- A description of how user-defined variables may be created in SADE. Debugger variables are useful for capturing values generated during a debugging session, evaluating expressions, and controlling debugger execution.

- Case sensitivity during symbol lookup. By default, case sensitivity is turned on when SADE performs symbol lookup.

As you read this chapter, keep in mind the distinction between those symbols "owned" by the debugger, and those "owned" by the program being debugged. A **debugger object** is either a command name, a predefined debugger variable, or a user-defined variable. These debugger objects are used to control debugger execution, or, in the case of user-defined variables, to save values during the debugging session. A **program symbol** refers to an address or an offset within the program being debugged: this includes procedure and function names as well as references to program variables. These program symbols refer to data that is used or altered by an executing program. Program symbols may be used to examine and modify program memory from within the debugger.

About Symbols

A symbol is a character or a combination of characters used to represent one of three things: an identifier, a numeric constant, or a string. These basic building blocks can be used to create all the various kinds of symbols known to the debugger. The SADE command language contains a number of helpful clues that make it possible to correctly identify symbols. These clues include the symbol search path, which is described in the following section.

A symbolic debugger is able to interpret a symbol within a program in much the same way that the Finder interprets an HFS pathname. The debugger first determines the proper scope for the symbol; that is, it determines where the symbol has meaning within the program. The debugger must then find the symbol, whether it represents a pathname, a program variable, or a register name. Some of the methods SADE uses to find symbols include the escape characters described in the section "Variable References", and the procedure qualifiers described in the section "Program Variable References".

Debugger symbol seach path

The concept of scope, which is important in programming languages, is just as important within a debugging environment. A symbol is only meaningful within the scope for which it is defined. For instance, local variable references are only meaningful within the procedure or function where they are defined. In addition, a symbol can't be interpreted by the debugger until a memory location has been allocated for it. This means that when a program is interrupted before a procedure containing symbol definitions has a chance to execute, the symbols within that procedure won't be usable by the debugger.

Within a command line, the first symbol found is evaluted to see if it is among the command names recognized by SADE. In all other cases, the search path for a debugger symbol can be summarized as shown below. If you don't explicitly state where to look for a variable reference, SADE tries to find the variable first as a debugger symbol, then as a user-program symbol, and finally as a system symbol.

Order of Symbol Lookup

1. Is it a debugger symbol?

   a. Is it a user-defined debugger variable parameter?

   b. Is it a local variable in a debugger proc?

   c. Is it a builtin SADE variable?

   d. Is it a global user-defined debugger variable?

2. Is it a program symbol? The program symbols are provided by compiler output, and are made accessible to SADE by using the Linker with the -SYM option.

   Once SADE has determined that a symbol is a program symbol, it uses the current name scope (CNS) information from within the program to ascertain:

   a. Is it a local program symbol?

   b. Is it a global program symbol?

3. Is it a system symbol, recognizable within the Macintosh Toolbox?

   a. Is it a system global variable?

   b. Is it a a toolbox routine name?

   c. Is it a register name?

Numeric Constants

This section describes how numeric constants behave within SADE. These constants may take the form of *decimal*, *hexadecimal*, *binary*, and *floating-point* numbers. They are commonly used when specifying an integer value, an address location, or a range expression. For more information on using numeric constants within expressions, see the section titled "Expressions".

The following paragraphs define the syntax for using numeric constants within SADE. The examples in this section show how each of the numeric types can be used.

Decimal — Decimal numbers are formed as a string of decimal digits (0-9). Values are treated as 32-bit (signed long word) quantities. Decimal values that exceed 32-bits are treated as floating point values.

Examples:

123	5	32
65535	123456	32768

Hexadecimal	Hexadecimal numbers are specified by a dollar sign ($) followed by a sequence of hexadecimal digits (0-9, A-F or a-f). A period may be used to group components of a hexadecimal number. Hexadecimal numbers are normally treated as 32-bit (left-padded with zeros if necessary) quantities. If *more* than eight digits are specified, the hexadecimal number is considered as a string (a zero will be padded on the *right* if there are an odd number of digits in the string). In effect, when the supplied hexadecimal value is too long, SADE performs a type coercion to a string.

Examples:

```
$123              $1A3c       $FFFF
$00123            $01a3C      $0FFFF
$1234.5678.9A.BC.DE           (equivalent to $123456789ABCDE)
$1234567890abcDEF             (a string)
```

Binary	Binary numbers are specified by a percent sign (%) followed by a sequence of binary digits (0-1). A period may be used to group components of a binary number. Binary numbers are treated as 32-bit (left-padded with zeros if necessary) quantities. If *more* than 32 digits are specified, the binary number is considered as a string (up to 7 zeros will be padded on the *right* to fill in the last byte in the string). In effect, when the supplied binary value is too long, SADE performs a type coercion to a string.

Examples:

```
%1010             %101        %011101
%1010.0011.1100.1111
```

Floating-point	Floating-point numbers are specified with a decimal point or exponent as described in the Apple Numerics Manual . Within SADE, floating-point numbers are represented as SANE 10-byte extended values.

Examples:

```
123.        123.4E-12    .123       NaN
Nan()       NAN(12)      INF
```

Both the hexadecimal and binary numeric constants may perform a type coercion to a string if the supplied value is larger than 32 bits. The value of a string created in this manner will depend on the context in which the resulting constant will be used. The string could still be evaluated, but each of the positions in the string would have an ASCII value, with the total value of the string being equal to the concatenated values of all the string elements.

Strings

Many of the SADE commands take a string as a parameter. For instance, filenames are always strings. Whenever you see that a command takes a *name, filename, program name* or a *str expression* for an argument, the argument that you supply must follow the rules for strings. These rules are described in this section.

A string is formally defined as a hexadecimal number consisting of more than 8 digits, a binary number of more than 32 digits, or a sequence of one or more ASCII characters (including blanks) enclosed in single (') or double (") quotation marks. You can also use quotation marks as part of a string. When you do this, two quotation marks (with the same value as the delimiting quotation marks) must be specified in the string for each quotation mark (with the same value). Strings are limited to a length of 254 characters.

Escaped characters may be specified in double quoted strings. These are represented by "\" or "∂" immediately followed by one to three decimal digits, or a one or two-digit hexadecimal number (\$xx), or one of the following single character reserved to represent certain non-graphic characters: \n (newline, \$0D), \t (tab, \$09), and \f (formfeed, \$0C). Any other character immediately following the backslash represents just that character, for example, \\ (backslash), \' (single quote), etc. You can also delineate the string with single quotes and use the double quotes as literals, or vice versa. Note this substitution does not occur when single quotes are used as delimiters.

Some examples of strings are:

```
'Hello'        'don''t'       "hello\0"      """" (one quote)
"Hello"        "don't"        "don""t"       "'"      """"""
'hello\0'      (escape characters not processed)
```

Based on the context in which you use a string, there are restrictions on how long a string can be and how it is treated. Strings fall into two categories:

A string *constant* used in an arithmetic expression (described later) which is to be used as a *numeric* value (for example, when combined with arithmetic operators) is limited to 4 characters. Such strings are treated as right-justified 32-bit signed values. Each of the characters in such a string is assigned its ASCII value, and the overall value of the string represents the concatenation of values of the string elements. For instance, the string "my" has the value (6D + 79).

String (constants) used in logical expressions (in relations) and strings used as search patterns may be up to 254 characters in length.

When executing a file, each command line is limited to a maximum of 254 characters. A maximum length string of 256 characters is too long when executed from a file, because the quotation marks used as delimiters are counted as part of the string length.

Identifiers

This section describes the various kinds of identifiers that can be used in SADE. These include variable references, predefined debugger variables, and register names. Identifiers are used by SADE as keywords in commands, labels, registers, and to reference program variables.

The first character of an identifier must be an uppercase or lowercase letter (A-Z, a-z), an underscore (_), or a percent sign (%). Subsequent characters can be letters, digits (0-9), underscores (_), dollar signs ($), number signs (#), percent signs (%), or "at" symbols (@). Other characters may be made a part of an identifier by quoting them with a preceeding "∂". A name may be any length, but only the first 63 characters are significant.

One thing to keep in mind when working with identifiers is the difference between qualified identifiers and unqualified identifers. A fully-qualified identifer contains all information needed for its use; SADE will not have to use the symbol search process, since the scope is completely defined in the identifer name. A partially-qualified identifier contains only part of the location information, and SADE deduces the rest from the current state of the debugging session. An unqualified identifier provides no information about its scope, so SADE assumes its proper place is in the first place it is found when searching.

To prevent ambiguity, you can use the following operators with identifiers:

∆identifier	system symbol
`identifier	program symbol

Use of these operators will keep SADE from having to go through the symbol search process. These operators are also described in the section titled "Variable References", since they are most useful for that class of identifiers.

Within the realm of identifiers, certain debugger symbols are predefined: these include command names and predefined debugger variables. Other symbols are created during the debugging session, subject to the rules described within this chapter. Command keywords are listed in Chapter 1. Register names, predefined debugger variables, and variable references are described in the sections below.

Setting Case Sensitivity

Identifiers corresponding to debugger objects are case-insensitive, whereas identifiers corresponding to program symbols may or may not be case-sensitive, depending on the conventions of the program source language. Each debugger window is either case sensitive or case insensitive. Source display windows have the same behavior as the source language displayed, using the filename conventions of MPW to determine the source language. The window's setting can be changed by using the Case command.

Setting case sensitivity in SADE is only significant when SADE is looking for symbols. The Case command works like a toggle switch to turn case sensitivity for symbol lookup on or off. By default, the case sensitivity for symbol lookup is turned on. This means that when SADE is presented with a symbol such as "ABC", it will use the lookup rules to find the first occurence of the ASCII symbol "ABC" in the largest scope currently in effect for the debugging session. If a lower-case symbol such as "abc" is found before "ABC", SADE will still continue looking for "ABC".

The format for the Case command is:

case { on | off }

In most cases, the default situation (case sensitivity is on), will provide the correct symbol lookup. If case sensitivity is turned off, the symbol lookup will halt at the first occurence of a selected string, whether the characters are upper or lower case. For instance, when searching for the symbol "ABC", SADE will look for "ABC" in the largest current scope; however, if it finds an "abc" first, the search will halt.

Variable references

This section provides an overview of the different types of variable references. Simple and structured variable reference types may be used within SADE using the guidelines explained below. In addition, this section tells you how SADE distinguishes between system, debugger, and system variables. Detailed information on predefined debugger variable, user-program variables, and register names is provided in the sections that follow.

A **simple variable reference** consists of only a single identifier. This would be a variable from within the program, such as "myVar". These variables may be of any type supported in the program being debugged.

A **structured variable reference** allows a simple variable identifier to be followed by the following selector operators:

.name	record or structure field selection
^	pointer dereference
...] or [n]...	array access

This allows SADE to use the some of the structured types common in high-level programming languages. For instance, you could use the array element "myArrayVar.[1]" as a variable reference within SADE.

The selector operators are allowed only if the type of the variable to which they are applied supports the specified selection operation. The result of such a structured variable reference is, of course, itself a variable reference to which further structure selector operators can be applied (if it refers to a variable of an appropriate structured type). This means you can reference nested structured types.

Variable references may be made to three different classes of variables—system, debugger or user-program variables. Each of these types are further described as follows:

- **System variables** refer to objects in the Macintosh ToolBox. These include register names, system global variables, and toolbox routine names. All the global variables and toolbox routines listed in *Inside Macintosh* can be accessed from within SADE. Register names recognized by SADE are listed in a following section.

- **Debugger variables** refer to either predefined debugger objects or user-defined debugger objects (variables defined by the user in the debugging session). The predefined debugger variables are listed below, while the user-defined variables are described in the next chapter.

- **User-program variables** refer to objects in the program being debugged. This type of variable is described in one of the sections which follow.

If you don't explicitly state where to look for a variable reference, SADE tries to find the variable first as a debugger symbol, then as a user-program symbol, and finally as a system symbol, as described in the section "Debugger Symbol Search Path". Since each of the three classes of variables represents a namespace created independently of the others, name collisions between the different classes of variables are possible. A variable in one namespace may "mask out" a variable with the same name in a namespace which is searched later.

To allow access to variables which might otherwise be "masked out", SADE supports two special characters or operators which can be used to force the class of a variable identifier to be either a system or user-program variable:

- **the system symbol escape character** ("Δ") may be prefixed to an identifier to indicate that it refers to a system variable. This allows a faster lookup than the normal symbol search process.
- **the user-program symbol escape character** ("`", backquote) may be prefixed to an identifier to indicate that it is a user-program variable

For example, assuming that there is a program symbol whose name is "pc" just like the system symbol representing the program counter, "Δpc" would refer to the debugger symbol and "`pc" or "pc"would refer to the program symbol.

Simple and structured variable references, as described above, are sufficient to access debugger and system variables. Global debugger variables and system variables exist in flat namespaces; that is, there are no hierarchical levels for these variables. User-program variables, on the other hand, exist in a hierarchical scheme that includes a program name, unit names, and procedure names. The methods for accessing user-program variable references are described later in this chapter.

Predefined debugger variables

The predefined debugger variables used within SADE establish a set of parameters for each debugging session. These debugging parameters influence how the debugger interprets a command and how it displays the results of command execution. At startup, SADE sets a default for each debugging parameter. The default values may be displayed by entering the variable name.

Many of the predefined debugger variables are read-only variables; that is, you can't change the value contained in the variable. The Arg and NArgs variables are used when supplying parameters to the current debugger proc. If you need more information on how to write a debugger proc, see Chapter 8, or the Proc command page in Part II.

The predefined debugger variables are described below with their default values:

Arg[n]	is the nth parameter of the current debugger proc. This is like an array variable for the debugger proc.

ActiveWindow	is a string containing the name of the topmost (i.e., active) SADE window. This is a read-only variable; it cannot be assigned to.
Date	is a string containing the the current date in the form "dd-mm-yy". This is a read-only variable; it cannot be assigned to.

DisAsmFormat is a string which contains letter "flags" that control the formatting of the Disasm command output. Each line of the disassembly output is divided into four fields, as described by the Disasm command.

The order and presence of the four fields are controlled by the "flag" letters *in* the DisasmFormat builtin variable. The initial value for DisAsmFormat is "OAXC". This means that the initial order of the fields shown during disassembly is offset, address, hex representation, and finally the assembly code. The following flags are allowed:

o	==>	display offset field in decimal.
O	==>	display offset field in hexadecimal
a, A	==>	display the address
x, X	==>	display the hex code representation
c	==>	truncate the assembly code if necessary to a uniform length
C	==>	show entire assembly code no matter how long
$	==>	prefix offset and/or address with a "$" (allowed only before O, a, and A flags)

The flags may be specified in any order, and blanks and tabs are ignored in the string. A flag specifying the presence of a field may not be repeated. Whenever the DisAsmFormat variable is used, at least one of the two flags "x"/"X" or "c"/"C" must be specified. If the assembly code field is specified as the last field, then "c" has the same meaning as "C"—the entire assembly code field is displayed. If the assembly code field is to be displayed before one of the other fields, then you have the option of either truncating the assembly field to a uniform length ("c") or showing it completely ("C").

The DisAsmFormat variable may also contain a "$" flag in front of the "O", "a", or "A" flags to generate a "$" character in front of the offset and/or address field values.

Exception is the exception number of the most recently encountered exception. This is a read-only variable; it cannot be assigned to. Some of its values (***more to come***) include:

2	Address Error
3	Bus Error
8	Instruction trace
9	Trap Break

13	Program interrupted at
14	Address Break

Inf is always equal to a SANE infinity. This is a read-only variable; it cannot be assigned to.

NArgs is the number of actual parameters specified for the current debugger proc. This is undefined when no debugger proc is in use.

ProcessId is the process identifier for the current target program. It marks the process that was suspended when the debugger was entered. When an exception occurs and SADE is entered, this variable is set to the process identifier of the process in which the error occured. It also changes when the Target command is used. This is a read-only variable; it cannot be assigned to.

WorksheetWindow is a string with the name of the SADE worksheet window This is a read-only variable; it cannot be assigned to.

User-program variable references

Each user-program variable may be identified by its place in a hierarchy that includes a program name, unit names, and procedure names. This hierarchical identification scheme is described in this section. Variable references, like identifiers, may be fully or partially qualified, depending on how much information is supplied in the variable name.

This section also describes the namespace for a program symbol. This namespace represents the result of various separate compilations or units linked together, and thus requires a means of identifying the program, unit and procedure where a variable may be found. The methods for using procedure and statement references, and for referencing variables outside the current procedure or function, are also described in this section.

A **fully-qualified variable reference** is one which identifies a variable with program name, unit and, if necessary, procedure qualification, as below:

\ *unit* [. *procedure*]* . *variable reference*

These components can be interpreted as follows:

backslash (\\) The backslash is used as the program-level or unit qualifier character, and should be used preceding the unit name.

[...]* The [...]* construct indicates that zero or more levels of procedure name qualification may be used: zero when accessing unit level variables, one when accessing first level procedures, and more when accessing nested procedures. (This is possible in Pascal but not in C).

unit When referring to the main program's variables the *unit* should be the program name (as opposed to the program file name). The unit names follow the same rules as do variable identifiers.

procedure	A procedure reference is used to refer to the starting code location of procedures. Among other things, a procedure reference may be used as a location for setting breakpoints. A procedure reference may be structured in much the same way as a variable reference, and may be fully-qualified or partly-qualified, as described below.
dot (.)	The conventional dot is used as the unit and procedure level qualification character.
variable reference	This can be constructed according to the rules for simple and structured variable references. (See the section "Variable References").

To correctly specify a procedure starting at the unit level, you must use the backslash preceding the unit name. Otherwise, SADE can't determine if the name refers to a unit or a procedure.

A program name surrounded by quotation marks, or a unit name preceded by a backslash, is interpreted at the global program level. Since program names may contain invalid identifier characters, such as spaces, they should be specified as a double-quoted string. Special characters may be escaped with the character escape symbol (∂).

A *procedure* reference is assumed to be local; if it can't be identified at the local level, SADE checks the global level. Any *variable reference* in a fully-qualified variable reference is interpreted at the local procedure level, unless explicitly specified otherwise. Fully-qualified variable references are similar to directory-relative absolute pathnames under HFS; the debugger doesn't have to do any extra work to find these symbols.

SADE also allows **partially-qualified variable references** to facilitate access to program symbols.This allows some or perhaps all of the initial qualification of a fully-qualified variable reference to be omitted, if it can be deduced from the current state of the debugging session. More specifically, if SADE is currently suspended in the middle of a program execution at some breakpoint, that point (the current Program Counter) defines a **current name scope** (referred to as CNS below), since the breakpoint identifies a program, unit, and perhaps procedure from which to begin looking up variable references.

If a partially-qualified reference omits only the program name and begins with the program level qualifier (the backslash), SADE treats the following identifier as the unit name, followed by all of the intervening qualification up to the variable name. This implies a straightforward lookup following the unit-procedure-variable hierarchy.

If a partially-qualified reference omits both the program name and the backslash (the program level qualifier), then SADE has to resolve a partial reference whose first component could be a unit, procedure, or variable name, depending on how much qualification was omitted. This requires a lookup relative to the CNS as defined by the current breakpoint. The first component of the variable reference (the leftmost identifier) will be looked up within the symbol table for the procedure or unit corresponding to the CNS. If the first component is found, then the remainder of the reference components are checked to see if they identify a valid reference path to some variable. If so, the variable reference has been resolved. If not, the initial match of the first component was invalid and the search continues. If the first component is not found in the symbol table for current procedure, the containing procedure's or unit's (if there is one) symbol table is searched.

If no match is found after reaching the unit level, the lookup in the user-program symbol table fails.

Program procedure and statement reference

A **procedure reference** can be used to refer to the starting code location of procedures. Procedure references may be used, for instance, in the setting of breakpoints. Procedure references follow the form of the variable references as described above, but omit the final variable qualification. They consist of a program name, unit name and one or more levels of procedure names (with some amount of the leading qualification omitted for partially qualified references).

Source program statements are identified by indices relative to a procedure (or function). These indices are identified by the compiler and are associated with locations in the text in source windows. A **statement reference** consists of a procedure reference followed by a dot delimiter followed by a reference of the form "(*expr*)", which refers to a particular statement index relative to the specified procedure. If the procedure reference is omitted and the "(*expr*)" form is used by itself, it is taken to refer to the current procedure (defined by the CNS as the point where execution has been suspended).

Referencing objects outside the stack activation

The notation for referencing program objects described in the previous sections is extended slightly to allow access to objects in other than the current stack activation—the current procedure or function call. An array-like specification (consisting of an expression in square brackets) may be inserted between the last procedure name and the beginning of a variable reference to indicate the n'th activation of that procedure from the top of the stack.

Register names

Register names are system symbols, and are not built into the SADE application. SADE uses these names to display the data your program places into the registers provided by the 68000 family of microprocessors. When you disassemble instructions, you can see what registers were used by a particular instruction. To use these register names from SADE, you should use the "Δ" prefix to distinguish them from program symbols. For example, to disassemble ten instructions starting at the program counter, use the PC register name as shown below:

```
disasm Δpc 10
```

The Disasm command is described in Chapter 5.

A list of the register names usable from SADE appears below:

D0..D7	Data registers
A0..A7	Address registers
CCR	Condition code register
SR	Status register
USP	User stack pointer

MSP	Master stack pointer
SP	Stack Pointer
SSP	System stack pointer
SFC	Source function code register
DFC	Destination function code register
CACR	Cache control register
VBR	Vector base register
CAAR	Cache address register
ISP	Interrupt stack pointer
PC	Program counter
FPCR	Floating-point control register
FPSR	Floating-point status register
FPIAR	Floating-point instruction address register
FP0..FP7	Floating-point data registers
TT0..TT1	MC68030 transparent translation registers

Expressions

This section describes the guidelines for using expressions in SADE. The detailed rules for operator precedence, expression operand base types, expression evaluation, type coercion, and address ranges are all contained in subsections. Builtin functions may also be used as expression elements; these are described in a following section.

Expressions are composed of either a single term or an arithmetic combination of terms. A term is either a named symbol, a constant, or a function call. Terms are combined by arithmetic, logical, shift, and relational operators. String terms may only be combined with relational operators.

Operator precedence

This section describes the operators used to form expressions within SADE. These operators are listed from highest precedence to lowest. Groupings within the table show operators of the same precedence; for instance, multiplication, division, and remainder all have the same precedence.

Highest Precedence

( )			Grouping by parentheses
@			Address of
†			Trap Expression
^			Pointer to
.			Qualifier
~			Bitwise ones complement
¬	NOT	!	Logical not

-			Unary negation
*			Multiplication
/	DIV	÷	Division
//	MOD		Remainder
+			Addition
-			Subtraction
>>			Shift right
<<			Shift left
=	==		Equal
<>	≠	!=	Not equal
<			Less than
>			Greater than
<=	≤		Less than or equal
>=	≥		Greater than or equal
&	AND		Bitwise and
&&			Logical and
|	OR		Bitwise or
||			Logical or
XOR	EOR		Bitwise exclusive or
:=			Size compatible assignment
<-			Arbitrary assignment
..			Range

Lowest Precedence

The rules for coding expressions are as follows:

Only the +, -, ~, and NOT (¬, !) operators are allowed at the start of an expression.

An expression may not contain two terms or operators in succession.

Subexpressions are designated by enclosing the subexpression in parentheses.

Parentheses may be nested to a maximum depth of 20.

An expression may not consist of more than 20 terms.

The keyword operators DIV, MOD, AND, OR, XOR, and NOT must be separated from identifier operands by at least one space.

The range operator (..) may only appear once in an expression.

Expression operand base types

This section describes the types of constants and variable references that may be used within SADE expressions. Operands (constants and variable references) used in expressions may only be combined with operators if they are one of the base types allowed by the debugger expressions. These base types are:

Boolean	A one byte Pascal boolean.
UnsignedByte	A byte with the value range 0 to 255.
Byte	A byte with the value range -128 to 127.
CChar	A byte with the value range 0 to 255.
PChar, PascalChar	A word with the range 0 to 255 (Pascal Char)
UnsignedWord, UnsignedShort	A word in the range 0 to 65,535.
Word, Short, Integer	A word in the range -32,768 to 32,767.
UnsignedLong, UnsignedInt	A long word in the range 0 to 4,294,967,295.
Long, Int, LongInt	A long word in the range -2,147,483,648 to 2,147,483,647.
Single, Float, Real	A IEEE floating point single-precision value (4 bytes).
Double	A IEEE floating point double-precision value (8 bytes).
Extended	A IEEE floating point extended-precision value (10 bytes).
Extended12	A IEEE floating point extended-precision value (12 bytes).
Comp[utational]	A SANE signed 8-byte integer.
CString	Up to 255 characters delimited by null byte.
PString, String, Str255	A length byte followed by up to 255 characters.
AsIsString	A string with no length byte and no delimiting null byte.

The rules for how these types are evaluated within expressions are discussed in the next section.

Evaluation of expressions

This section describes how a single-term or multi-term expression is evaluated by SADE. A single-term expression is represented by a single symbol, and takes on the value represented by the symbol (the value associated with the name, the constant, or string). If a symbol represents an array or record structure, the "value" of the expression is the entire array or record structure.

A multi-term expression consists of two or more operands. Multi-term expressions are reduced to a single value according to the following set of rules:

- Each signed integer operand is converted to a 32-bit signed value.
- Each unsigned integer operand is converted to a 32-bit unsigned value.
- Each floating point and computational value is converted to a 10-byte extended value.
- When a binary operator combines two *integer* operands, both operands are treated as unsigned if either is unsigned. The result is then treated as a 32-bit unsigned value.
- If both integer operands combined with a binary operator are signed, then the result is a signed 32-bit value.
- If either operand is a floating point value, then the other operand is converted to floating point extended and the result is extended.
- Integer division by zero yields zero as the result. Floating point division by zero yields infinity, except for zero divided by zero, which yields a NaN.
- Operations are performed from left to right, following the precedence indicated in the operator table above. Assignment operators are performed right to left.
- A parenthesized subexpression is reduced to a single value. The resulting value is then used in computing the final value of the expression.
- When parenthesized subexpressions are nested, the innermost subexpression is evaluated first.
- Integer division always yields an integer result; any fractional portion of the result is dropped.
- The logical operators NOT (¬, !), = (==), <> (≠, !=), >, <, <= (≤), >= (≥), &&, || evaluate to the value 1 (true), and the value 0 (false). Comparison is algebraic, except when two character strings are compared.
- The shifting operators << and >> shift the left operand by the number of bits specified in the right operand. Zeros are shifted into vacated bit positions. Bits shifted out are lost.

- The assignment (:=, <-), pointer (^), trap (†), and address (@) operators are special operators with meanings unique to SADE. They are discussed separately in the following sections.

The Assignment operator

The assignment operator in SADE is treated as a binary operator. As such it may be embedded in a more complex expression to capture intermediate results. The assignment operator is the only operator which evaluates right to left. Thus an expression of the form,

```
        a := b := c := d
```

is evaluated as if it had been written,

```
        a := (b := (c := d))
```

The left operand of an assignment must be a variable reference. For a integer reference, the right operand is saved in the specified variable. For a floating point assignment, the right operand is converted to extended before the assignment if necessary. For string, record, or array assignments, the left variable must be compatible with the right operand, and no other operators may be combined with the assignment.

Compatibility between operands in SADE is defined as it is in Pascal for real and integer. For structured data, compatible operands are defined as having the same aggregate size. A second operator is provided for *arbitrary* assignment, namely <-. Using this assignment operator, you may assign any tupe to any other type, regardless of size. For arbitrary assignments, the size of the operand on the right hand side of the operator is used to determine the number of bytes to move. This operator may be used, for example, to patch memory. It may not be used to assign to debugger variables.

The Pointer operator

When the pointer symbol is used as an operator, it *follows* an expression term. When the term is a variable reference(such as x^), the pointer operator indicates an indirect reference through the variable, and the type of the term is determined by the type associated with the pointer variable reference. When the term is a (sub)expression, the pointer operator indicates an indirect reference through the address represented by that (sub)expression. The type in this case is assumed to be a *pointer* to a longint. Type coercion (described below) may be used to treat the reference as some other type. Note that (sub)expressions used in conjunction with pointer operators frequently use the address operator. For example, one may write WindowPtr((@X+Next+4)^).

The Address operator

A pointer to a variable (an address) can be generated with the address operator (@). The address operator is a unary operator taking a variable reference as the operand. The type of the value is considered as a pointer to the type of the variable.

The Trap operator

An expression whose value is a trap can be created using the trap operator (†). The trap operator is a unary operator taking an expression element or a parenthesized expression as the operand. Such trap expressions are used with breakpointing commands to distinguish trap breakpoints from normal address breakpoints.

Type coercion

Names of known types may be used in a function-like notation (as in Pascal) to perform type coercions on expressions. The type of the object being coerced will be changed as long as there is a reasonable way to interpret and perform the coercion. The syntax is as follows:

type (expr)

The names of types may be either simple type names, or may be qualified as described in the section on program variable references. This allows SADE to distinguish type names defined in more than one scope.

Additionally, the type specification may be preceeded by the pointer operator ("^") to indicate coercion to a pointer to the specified type. This is an extension of the Pascal notation which allows type declarations such as "^integer".

The following examples illustrate how the type coercion mechanism works in conjunction with indirect memory references.

comp(10)	converts the number 10 to the comp (computational) type
comp(10^)	converts the long at location 10 to the comp type
^comp(10)	identifies 10 as a pointer to a comp
^comp(10)^	returns the comp at location 10

Ranges

Ranges of addresses or values can be expressed by a pair of expressions (the low and high ends of the range) separated by the range operator ("..", i.e., two dots). The syntax is as follows:

expr .. expr

Neither expression used to designate a range can be a floating point value. If one end of the range expression is a trap number, both must be. Trap ranges are expressed in a similar fashion using trap expressions on both sides of the range operator, for example, "†$A000..†$AFFF".

Builtin Functions

Builtin SADE functions provide a wide range of useful services. These functions can be used to return strings or numeric values, address information, or a TickCount value. The AddrToSource and SourceToAddr functions provide support for source-level debugging, while the Confirm and Request functions provide a way to communicate with user dialog boxes.

Each of these builtin functions may be used as part of an expression. For instance, you can use the Concat function, which returns a string expression, in any SADE command that takes a string as an argument. The builtin functions, their arguments, and the values they return, are described individually in the sections which follow.

AddrToSource

The AddrToSource function is used to display a read-only source window. If you have examined the contents of the initial SADEStartup file on the release disk, you'll see the AddrToSource function used within the SetSourceDisplay debugger proc. To use the AddrToSource function, supply an address expression and an optional boolean value as arguments:

AddrToSource (*addr-expr*[, *bool-expr*])

The function returns a boolean value, which will be a "1" (true) or a "0" (false), to indicate whether or not it was able to display the source window (a source file) corresponding to a particular code address. If the source file is displayed, the program statement corresponding to that address will be highlighted in the source window.

The optional boolean expression controls whether the window is brought up as the topmost (active) window. The default (when the boolean is false or omitted) is for the source window to be displayed behind the topmost window (from which the command with the AddrToSource function was likely to have been issued). Remember that the source windows brought up by the AddrToSource function are read-only windows. These windows have a "no-write" icon (a pencil with a slash through it) in the lower left corner of the window.

Concat

The Concat function is used to concatenate a series of string expressions. To use this function, supply the string expressions as arguments:

Concat ([*string-expr*,...])

The function returns a string whose value is the concatenation of the argument strings. If no arguments are specified, a null string is returned.

Confirm

The Confirm function is used to convey a user's response to the contents of a confirmation dialog box. The first argument string passed to the Confirm function will be displayed in the dialog box. To use the Confirm function, supply the display string and an optional boolean expression as follows:

Confirm (*string-expr* [, *bool-expr*])

The function returns a numeric value indicating the user's response to a confirmation dialog box. If the optional second boolean argument is omitted or equal to zero, an OK/Cancel dialog is presented. A response of "1" (true) is returned when the user selects the OK button, and a "0" (false) is returned is the user selects Cancel. If the boolean argument is non-zero, a Yes/No/Cancel dialog is presented, and "1" (yes), "0" (No), and "–1" (Cancel) are returned.

Copy

The Copy function can be used to copy all or a portion of a string. To use the Copy function, supply the arguments described below:

Copy (*string-expr*, *expr>*, *expr*)

The function returns the substring of the first string argument, starting at the character specified by the second argument. The length of the substring is supplied by the third argument or by the end of the string.

Find

The Find function looks for a target pattern, and returns either an address, a number value, or a zero, depending on what arguments are used. To use the Find function, supply the arguments as described below:

Find (*target-expr*, *addr-expr*, *length-expr* [, *count-expr*])

If the *count* argument is omitted, Find returns the address of the target pattern in the range specified by the address and length expressions. It will return zero if the target pattern vas not found. If the count is present and equal to zero, Find returns the number of occurences of the target pattern in the specified range. If the count is non-zero, Find returns the address of the count'th occurence of the pattern in the range.

Length

The Length function simply tells you the length of a string. To use it, supply the string as an argument:

Length (*string-expr*)

The function returns an numeric value indicating the length of the string you specified.

NaN

The NaN function is used to convert a specified value into a SANE NaN. Its format is:

NaN (*expr*)

The function returns a SANE NaN.

Request

The Request function returns a string after displaying a request dialog box. The first string argument is displayed in the dialog box as the request message. The second, optional string argument specifies a default string to present in the request box. To use the Request function, specify these arguments as shown below:

Request (*string-expr* [, *string-expr*])

If the user cancels the dialog, the string '_CANCEL_' will be returned. Otherwise, the request specified by the user is returned as a string.

SizeOf

The SizeOf function is used to calculate how many bytes of storage will be needed for a parameter, variable, or type. To specify a particular parameter, variable, or type, use the following format:

SizeOf (*parameter* | *variable* | *type*)

The function returns a the number of bytes needed to store the specified parameter, variable, or type.

SourceToAddr

To use the SourceToAddr function, you supply a string expression containing the name of a window that currently displays a source file. This does not have to be the active window, but can be any source window in which you have selected one or more statements. The function will calculate the address of the instruction that corresponds to the selected source statement, and displays the address location symbolically. The format of the function is:

SourceToAddr (*string-expr* [, *expr*])

Normally, SourceToAddr returns an symbolic expression for the address corresponding to the selection from the source file. A value of zero is returned if SADE wasn't able to determine what address the window selection represented. The optional parameter supplies an error message telling you why SADE wasn't able to return the address. This message will be displayed only if a value of zero is returned by the first parameter.

Timer

The Timer function uses the global variable TickCount to provide timing-related functions. To use this funtion, supply the following parameters:

Timer ([*expr* [, *bool-expr*]])

If you use the Timer function without any arguments, the current TickCount is returned. If one argument is specified, the value (TickCount - n) will be returned; this is the difference between the current TickCount and the value you specified (usually a previous value of TickCount). If the second boolean argument is specified and is non-zero, this difference will be returned as a string of the form "sss.hh", representing seconds and hundreths of a second. If the boolean argument is zero, then the function returns the same value as in the one-argument case.

TypeOf

When you need to return the type of an expression, you can use the TypeOf function. Its format is:

TypeOf (*expr*)

The function returns a string containing the name of the type of the expression value. If SADE does not know the name of the type, then a string of the form "Type #n" will be returned, where "n" is SADE's internal index for the type.

Undef

The Undef function is used to determine if a SADE parameter or variable is initialized. Its format is:

Undef (*parameter* | *variable*)

The function returns a "1" (true) if the parameter or variable is initialized, and returns a "0" (false) if it is not.

Where

The Where function is useful when you want to get a symbolic representation of an address. Its format is:

where (*addr-expr*)

The function returns a string expression.

SADE Command Language

An understanding of the SADE command language allows you to take advantage of SADE's many powerful features. The SADE command language was designed to be programmable and extensible. The debugger language itself uses English-like keywords so that command scripts and user-defined debugging procedures can be readable and intelligible. This section describes how the SADE command interpreter processes commands.

A **command** is the unit of execution for SADE's command interpreter. Each command is composed of a sequence of keywords, names, expressions, operators, and other special characters, terminated by the end of the line. Multiple commands may appear on the same line by using a semicolon (";") as a **command separator**. If a command is incomplete by the end of a line, the line can be continued by escaping the carriage return with the **character escape** ("∂").

A command that consists solely of an expression (other than an assignment) is evaluated and the result is written to standard output. For instance, if you enter the debugger variable processID, SADE will return a value as shown below:

```
processID
3
```

This mechanism is equivalent to a Printf command with a default format specification. For the example shown above, it is the same as entering:

```
printf "%t" processID
```

Since assignments are simply one form of expression, a command consisting only of an assignment expression won't have its value written to standard out. For instance, if you just assign processID to a variable as shown below, SADE won't immediately return the value; you'd have to use a printf statement to print the value of x.

```
x := processID
```

All other commands begin with a command keyword. The syntax of keyword commands is discussed throughout this manual; the command pages in Part II provide a handy way to locate commands alphabetically if you need to quickly check command syntax.

Comments are allowed anywhere in debugger commands, except within strings, and are delimited by "#" and the end of the line.

If you want to customize the SADE interface so that you can perform debugging tasks with fewer keystrokes, the debugger language supports a mechanism for abbreviations. Through use of the Macro command, you can set up a terse, short-named command set if desired. The SADEStartup file contains a number of macros that allow you to use Macsbug-style terminology for some SADE commands. You may want to design your own set of macros and place them in a UserStartup file; more information on this topic is contained in Chapter 7, "Customizing the Debugging Environment".

Command execution

This section briefly describes three of the ways to execute SADE commands: by pressing the Enter key, by selecting a menu item, or by using the Execute command with the name of a command file.

When a text window has been opened by SADE, any portion of text within the window can be selected and highlighted. The selected text can then be executed by pressing the Enter key. If no text is selected, the Enter key will execute any commands in the line containing the insertion caret.

Commands can also be associated with menu items in SADE's menus. Selecting such a menu item causes the associated commands to be executed. If a command key is assocated with a menu item, the command key will also cause the commands to be executed.

Finally, commands saved in a file can be executed using the Execute command, as described in Chapter 7. The commands in a special file, named "SADE Startup", are executed automatically when SADE is started.

Debugger Output Files

This section describes how SADE creates and manages debugger output—that is, the output from debugger commands. Related topics include debugger error output and symbolic display of command output; these are described in subsequent sections. Note that in these sections, the terms "window" and "file" are often used interchangeably.

All debugger output is directed to a text file, designated as standard output, or to the SADE WorkSheet window. The default standard output is the same window from which the commands were entered—referred to here as the command window. When output is written to the command window, the output immediately follows the currently selected text. If no text was selected, the output will begin on the next available line.

Redirected command output from SADE can replace, or be appended to, the contents of a window (file) using the Redirect command. The syntax of the Redirect command was described in Chapter 2. The simplest way to use redirection is to replace the contents of the named file. Using the **append** option adds command output to the end of the named file.

You may also nest Redirect commands, and send command output to as many as 10 different files. When output is redirected, SADE keeps a record of the name of the previous output file, so that when the commands whose output was redirected are completed, output can revert to the remembered file. Redirecting output to *filename*.§ will append the command output to the currently highlighted selection in the specified file.

SADE maintains the names of output files used with the Redirect command as a last-in first-out queue. If you use the **pop** parameter, or if you use no parameters at all with the Redirect command, SADE redirects the command output to the file at the head of the queue. If **all** or **pop all** is specified, standard output is redirected to the SADE WorkSheet.

Debugger error output

SADE doesn't include any standard error or diagnostic files. Any errors generated in the course of SADE command execution fall into one of two categories: parse-time errors, generated when command syntax is incorrect, and run-time errors, which happen when the commands are actually executed. Parse-time errors are typically written out to the same worksheet window used to enter the commands.

If a run-time error occurs while SADE commands are executing, the commands will be aborted, and any error messages will be written to the command window. If output was redirected to another window, the redirection is undone; in these cases SADE performs an implicit **pop all** for any redirected files.

Symbol Display

By default, a SADE window will display debugger command output symbolically. The Symbols command can be used to change the default behavior; it works like a toggle switch, turning symbol display off or on. Its format is:

```
        symbols [ {on|off} ]
```

Specifying symbols off will disable attempts to display symbolic representations of debugger command outputs. Turning off symbols might be desirable for speed, but in most cases you will want to use the default setting of symbols on. If the Symbols command is used with no argument, the current symbol setting is written to standard output.

Debugger Variables

Variable references may be made to three different classes of variables--system, debugger or user-program variables. The following section describes user-defined debugger variables, which are variables defined by the user in the debugging session. Predefined debugger variables are listed in a previous section. Rules which apply to all three types of variables can also be found in the section "Variable References".

User-defined variable references

A user can define a debugger variable on one of two levels:

- If a variable is declared within the debugger, but outside of a user-defined proc or func, then its scope is automatically global. This is known as the outer level.
- If a variable is declared at the inner level, inside of a user-defined proc or func, it is usually a local variable for that proc or func.

Global variables are those known both at the outer level and inside of user-defined procs and funcs. If a variable is declared at the outer level, the variable will be global in scope. If a variable is declared inside a proc or func, then its scope is local unless the define command is used with the **global** keyword. If a global and a local variable exist with the same name, then the local symbol overrides the global.

If you redefine a global variable, the new definition replaces the previous definition with one exception: If the definition is within a user-defined proc or func, and the new definition matches the existing definition, the existing definition is retained. For example, a global variable definition within a user-defined proc or func creates the variable the first time the procedure is invoked; subsequent invocations can make use of the value left in the variable by the preceding invocation.

The define command

The define command is used to create variables of arbitrary types for use in capturing values, evaluating expressions, and in controlling debugger execution. Its command syntax is

define [**global**] *var declaration* , ...

where a *var declaration* has the form

name [[*dimension*]] [:= *init value*]

where *dimension* is an *expr* and *init value* is either an *expr* for the initial value of simple types or a list of the following form for array variables:

([*expr* **of**] *init value* , ...)

where the optional **of** clause allows for replication of a value or set of values.

The **define** command may not appear inside a loop or conditional construct.

Debugger variables must be defined before they are used. A debugger variable declaration identifies the name, scope, and (optionally) the initial value of the variable. One or more variables may be declared in a single define command by having one or more *var declarations* separated by commas.

The *name* in a *var declaration* must follow the rules for valid debugger identifiers, and must be unique in the current debugger scope, such as a proc or func, unless declared global.

The *name* may optionally be followed by an array specifier--a dimension *expr* enclosed in brackets.

Debugger variables are dynamically typed, that is, their type is determined on assignment (and may be changed by new assignments). The only type information supplied at definition time is whether the variable is an array or a scalar. Debugger array variables may contain a heterogeneous set of values; that is, the elements may contain values of different types.

A initial value for simple types may optionally be specified by an *expr* following an assignment operator. If the item being declared is an array the fields of the type being initialized will control the assignment of the values from the list of initial values.

Debugger variables, once defined, may be referenced in expressions in debugger . The value of a debugger variable can be modified using the assignment expression operator (:=). If you wish, you can include the definition and the assignment on the same command line, such as:

```
define X := 5
```

However, it is not necessary to assign a type to a debugger variable when it is defined. The type of a debugger variable is set when a value is actually assigned. The debugger variable must have the same dimension as the type assigned to it. Components of structured debugger variables may be accessed using the selector operators described in the section on "Variable References".

The Undefine Command

The Undefine command simply removes a previously-declared variable. While the value of a variable can be changed merely by reassigning it, there may be times when you will want to completely remove the variable instead. This command may also be used with proc, func, or macro definitions. Its format is:

undefine *identifier*,

The *identifier* used may be any valid SADE identifier.

Chapter 4
Basic Debugger Operations

This chapter explains some of the most basic debugger operations available in SADE. After you've begun a debugging session, you need to know how to locate and control your application's files. The basic application control commands are included in this chapter, while more complex commands for controlling program execution are described in Chapter 5, "Program Control."

Among the basic functions you expect from any debugger is the ability to display information from within your program. You will often begin by displaying the last intruction executed before you entered the debugger. You may also want to display areas such as the application heap, which contains the bulk of your code, or the stack, which contains information about procedures that were called. SADE gives you a wide variety of ways to display program information; some of these display options include:

- produce formatted output to the current file or window. The value to be displayed is used with a type specifier, and the field width, justification, and precision are all controllable.
- display the location of address or trap breakpoints, using a symbolic represention when available
- display local symbols within a program, including the local procedure, parent procedure, local variables, and type information
- display process information at the time the application was suspended
- disassemble instructions and display the offset, address, hex representation, and assembly code for each instruction
- dump memory and display each instruction in hexadecimal and ASCII characters, according to byte, word, or long grouping
- display stack frames for the current target application
- display the application heap zone, including information about block type, master pointers, and whether the block is locked or nonrelocatable. The heap can also be checked for consistency.
- display resource information, including the location of each resouce map, and a list of the instances of each resource type. You can also check the resource map for consistency.
- search through memory for a numeric or string expression, and display the result

The "Eventlog" sample program is used as a basis for the displays seen in this chapter. In particular, the sample program was halted within a procedure named "DisplayText". See Figure 2-2 to see the SADE window and source display for this sample.

Locating and Controlling Program Files

This section deals with program files on the "outer" level—the commands described here deal with the application only as a filename. There are three kinds of files that are important when you're debugging. The first is the application itself, which is an executable file of type 'APPL'. The other important files are the symbol file and the source files for your application. These files can reside anywhere on your system, so SADE needs a way to locate them.

The sections that follow divide these application control commands into two groups—those that locate application files, and those that control your application from this outer level.

Locating your Files

As described in Chapter 2, your SADE directory can be used to store files connected with SADE's operations, such as the SADEStartup file, the Worksheet file, and any other files you may create during a debugging session. However, you wouldn't normally place your application files in the SADE directory. In most cases, you already have a one or more directories for your source files, object files, symbol file, and the executable application.

To use SADE with your application files, you must provide enough information so the debugger can locate them. The Directory, Target, and Sourcepath commands, described below, allow SADE to locate application files within various directories.

Directory Command

When you first enter SADE, the default directory is the directory where SADE resides. The Directory command sets the default directory for all SADE operations to the specified directory. You can use this command to specify the directory where the application resides. Its format is:

directory [*directoryname*]

The directory name is a string expression. For example, if you want to specify a directory named "Samples", use the Directory command as follows:

```
directory ":Samples"
```

If you use the Directory command with no argument, the current default directory is written to standard output.

Target Command

When you are running an application program and then break to the debugger, the program that was interrupted becomes the **target program** for subsequent SADE commands and symbolic references. The target program and its symbol file may reside anywhere on your system; on your system, you may have placed the target program and the symbol file in separate directories. When using the name of the target program in SADE commands, remember to use a filename relative to the current directory. If you need to change the current directory, use the Directory commmand.

The Target command allows you to specify a target program and its associated symbol file. The syntax for the Target command is:

target *progname* [**using** *symbolfilename*]

The selected target program will be used as the object for all following SADE commands and symbolic references. The optional **using** parameter may be used if the name of the symbol file for the program is not *progname*.sym in the same directory as the application. This allows SADE to find the symbol file even if it's in another directory.

For instance, to specify the "Eventlog" program as the target program, and use a symbol file that's in different directory, enter:

```
target ":Eventlog" using ":mysymboldir:eventlog.sym"
```

If you don't include the **using** parameter, SADE will expect to find the symbol file "Eventlog.SYM" in the same directory as the "Eventlog" program.

SourcePath Command

SADE permits source level debugging when the application's source files are available. This capability is implemented through the AddrToSource function, described in Chapter 3. To allow SADE to access an application's source files, you must provide a search path so that SADE can find them.

The SourcePath command is used to specify the search path used for source file display. Its format is:

sourcepath [[**add** | **del[ete]**] *directoryname* , ...]

The specified *directoryname* indicates where the AddrToSource function should look to find files for source display. You can use a list of directory names to allow the use of source files in more than one directory. The **add** and **delete** options allow particular directories to be added (to the end) and removed from the search path

If you use the command with no arguments, the current search path of directories is written to standard output.

As an example, let's assume that the source files for the Eventlog sample program reside in two directories: srcdir and myotherdir. To ensure that these directories will be searched for source file display, enter the SourcePath command with the following arguments:

```
sourcepath ":scrdir", ":myotherdir"
```

Additional examples of source level debugging techniques can be found in Chapter 8.

Basic Application Control

You can launch or kill an application program by name from a SADE worksheet, without returning to the Finder. The Launch, and Kill commands are described below. Note that for these commands, filenames must always be relative to the current directory.

The commands presented in this section deal with the application program as a filename representing the program as a whole. The program control commands presented in Chapter 5 provide a means to get inside an executing application program, and control its execution within SADE.

Launch Command

The Launch command launches the application you specify. The application will run normally until interrupted by the NMI key, or some other method. This command does nothing if the filetype of the specified file is not 'APPL'. Its syntax is:

launch *filename*

where *filename* is a string expression. For instance, to launch the "Eventlog" program, enter:

```
launch ":Eventlog"
```

Refer back to Figure 2-2 for an example of an application program that has been launched and then halted.

Kill Command

The Kill command halts the execution of the application or tool you specify. You can only kill processes that are already suspended. Its syntax is:

kill *filename*

where *filename* is a string expression. For instance, to kill the program "Eventlog", enter:

```
kill ":Eventlog"
```

Caution: this command can be dangerous if your application is killed without having a chance to perform its normal exit routines. It's generally preferable to resume execution and then quit from the application in the usual way.

Displaying Program Information

This section provides a description of how to display a variety of information from your program. The sample program shown in this section is the "Eventlog" program, which you learned how to launch in the preceding section. The displays shown in the sections below assume that the program was launched, and then halted at the beginning of the DisplayText procedure.

The simplest way to display the value of a program symbol is to enter the symbol name in the SADE worksheet. SADE will evaluate the symbol and display its type, if possible. The debugger output is always in the same type as the symbol; the default radix for numeric types is decimal. Address values (pointers) are displayed in hexadecimal. For example, to display the local variable "len" (type long) from the DisplayText procedure, just enter:

```
len
3326014
```

The Where builtin function is useful for displaying the location of a program or system symbol. For example, to display the current location of the program counter, simply enter:

```
where(pc)
DisplayText.(0)
```

If you are familiar with window records, you might like to display something a little more complicated, as in this example from the SADE Worksheet:

```
^WindowRecord(windowList)^
```

This will display the entire WindowRecord data structure for the current FrontWindow (pointed to by the windowList). Or you can display just a part of the WindowRecord, as shown below:

```
^WindowRecord.port.portBits.bounds(windowList)^
RECORD
  top: 0;
  left: -1648;
  bottom: 32;
  right: 128;
END
```

The List command can also be used to display program symbols, as well as other program information pertaining to processes, breakpoints, and tracepoints. See the section titled "Listing Program Information" later in this chapter. For other displays, such as code diassembly, memory dumps, and stack frames, see the section on "Special-purpose Displays".

Another way to display a program location is to use the Printf command with the symbol for that location. The Printf command produces formatted output, and includes many optional parameters. A complete description of the Printf command is included in the following section, with a few brief examples.

Formatted Displays

The Printf command places formatted output on the current output file or window. Its many optional parameters can be used to specify what to display and how to format the display. The value to be displayed is used with a type specifier, and the field width, justification, and precision are all controllable. The Printf command syntax is:

printf [*format* [, *arg*] ...]

or

printf ([*format* [, *arg*] ...])

The *format* parameter is a string that specifies the format of the output. The *arg* parameters are used to specify the values to be output. If no *format* and *arg* parameters are specified, any buffered output is displayed.

The format string contains characters to be copied "as is" to the output and conversion specifications. Each of the format string characters applies to zero or more arg parameters. If the format is exhausted while arg parameters remain, the extra arg parameters are ignored. If there are insufficient arg parameters called for by the format, then the rest of the format string is ignored.

For example, to display in hexadecimal the address of the next instruction to be executed, enter the Printf command with the name of the program counter register (PC) as a parameter:

```
printf "%.8X\n",pc
$00378B7A
```

Here the format string is ""%.8X\n"", while the arg parameter is "pc".

To display the address of the current heap, use Printf with the global symbol theZone:

```
printf "$%.8X\n",theZone
$00378420
```

In this example, ""$%.8X\n"" is the format string, while the arg parameter is "theZone".

The optional parameters for Printf are described in the following section.

Printf Optional Parameters

The conversion specification is distinguished from characters to be copied as is in the format string by preceding it with a % character followed by a sequence of fields which describe how to format a arg value:

% [flags] [width] [precision] op

flags An optional sequence of characters which modify the meaning of the main conversion specification:

- - Left-justify within the field width rather than right-justify if the converted value has fewer characters than the specified minimum field width.

- + Always generate a "+" or "-" sign when converting signed arg values. Note, that negative values are always preceded by a "-" regardless of whether the "+" flag is specified.

- space Generate a space for positive values and "-" for negative values. This space is independent of any padding used to left or right-justify the value. The "+" flag has precedence over the space flag.

- # Modify the main conversion operation. The modifications performed are described in conjunction with the relevant main conversion operations discussed later.

width An optional *minimum* field width, specified as a decimal integer constant (that doesn't begin with a "0") or an "*". In the latter case a corresponding arg parameter specifies the minimum field width. If the converted value has fewer characters than the width, it will be padded to the width on the left (default) or right (if the "-" flag is specified) with spaces (default). If the converted value has more characters than the width, the width is increased to accommodate it. For %t conversions, the width specifies the minimum width to reserve for RECORD type field names.

precision The optional precision is specified as a "." followed by an *optional* decimal integer or as an "*". In the latter case a corresponding arg parameter specifies the repetition count. If the decimal integer or "*" following the "." is omitted, the precision is assumed to be 0. Precision is used to control the number of digits to be output for numeric conversions or characters for string conversions. Omitting the precision has a default value which is a function of the main conversion to be performed.

op The required main conversion operation specified as one of the following single characters:

- d The corresponding arg parameter is converted to a *signed* decimal value (floating point values will be truncated).

  precision The precision specifies the minimum number of digits to appear. If the value can be represented with fewer digits, leading zeros are added up to the specified precision. The result of converting a 0 value with a precision of 0 is a null. The default precision is 1.

flags	-	left-justify
	+	explicit "+" or "-"

space	space for positive value
#	*ignored*

u The corresponding arg parameter is converted to an *unsigned* decimal value (floating point values will be truncated).

precision The precision specifies the minimum number of digits to appear. If the value can be represented with fewer digits, leading zeros are added up to the specified precision. The result of converting a 0 value with a precision of 0 is a null. The default precision is 1.

flags		
	-	left-justify
	+	*ignored*
	space	*ignored*
	#	*ignored*

X The corresponding arg parameter is converted to an *unsigned hexadecimal* value. The number of bytes converted is a function of the arg's type. The letters abcdef are used for x conversion and ABCDEF are used for X conversion.

precision The precision specifies the minimum number of digits to appear. If the value can be represented with fewer digits, leading zeros are added up to the specified precision. The result of converting a 0 value with a precision of 0 is a null. The default precision is 1.

flags		
	-	left-justify
	+	*ignored*
	space	*ignored*
	#	prefix converted value with a "$"

b The corresponding arg parameter is converted to an *unsigned binary* value. The number of bytes converted is a function of the arg's type.

precision The precision specifies the minimum number of digits to appear. If the value can be represented with fewer digits, leading zeros are added up to the specified precision. The result of converting a 0 value with a precision of 0 is a null. The default precision is 1.

flags		
	-	left-justify
	+	*ignored*
	space	*ignored*
	#	*ignored*

o The corresponding arg parameter is converted to an *unsigned octal* value. The number of bytes converted is a function of the arg's type.

precision The precision specifies the minimum number of digits to appear. If the value can be represented with fewer digits, leading zeros are added up to the specified precision. The result of converting a 0 value with a precision of 0 is a null. The default precision is 1.

flags	-	left-justify
	+	*ignored*
	space	*ignored*
	#	prefix converted value with a "0"

f The corresponding arg parameter is converted to a signed decimal *floating point* value. The value is converted to the form "[-]ddd.ddd", "[-]INF", or "[-]NAN(ddd)" (where ddd is the NAN code) depending on the value.

precision The precision specifies the number of digits after the decimal point. If the precision is 0, no decimal point appears (which can be overridden with the "#" flag). The default precision is 6.

flags	-	left-justify
	+	explicit "+" or "-"
	space	space for positive value
	#	force decimal point in the case where no digits follow it

E The corresponding arg parameter is converted to a signed decimal *floating point* value. The value is converted to the form "[-]d.ddde±dd" (for e conversion), "[-]d.dddE±dd" (for E conversion), "[-]INF", or "[-]NAN(ddd)" (where ddd is the NAN code) depending on the value. The exponent will always contain at least two digits.

precision The precision specifies the number of digits after the decimal point. If the precision is 0, no decimal point appears (which can be overridden with the "#" flag). The default precision is 6.

flags	-	left-justify
	+	explicit "+" or "-"
	space	space for positive value
	#	force decimal point in the case where no digits follow it

G The corresponding arg parameter is converted to a signed decimal *floating point* value. The value is converted using f *or* e conversion (or in the style f or E conversion when G is specified). The form of conversion depends on the value being converted; e or E conversion is performed only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeros are removed from the result (which can be overridden with the "#" flag). A decimal point appears only if it is followed by a digit (which can be overridden with the "#" flag)

precision The precision specifies the *total* number of significant digits. If the precision is less than 1, then 1 is assumed. The default precision is 6.

flags	-	left-justify
	+	explicit "+" or "-"
	space	space for positive value
	#	force decimal point in the case where no digits follow it and keep trailing zeros

c The corresponding arg parameter is converted to a character (the value mod 256 is used).

precision *ignored*

flags	-	*ignored*
	+	*ignored*
	space	*ignored*
	#	*ignored*

s Unless the "#" flag is used, the corresponding arg parameter must be a string type (or a pointer) and the value is copied to the output as is. C strings and as is (Pascal packed array of char) strings are copied until a null is encountered (for C strings) or the number of characters specified at the precision is reached. Pascal strings may be processed if the type of the arg is a Pascal string. When the "#" flag is used, the corresponding parameter is treated as an unsigned long, and printed as if it contains 4 characters.

precision The precision specifies the maximum number of characters to output. The default precision is assumed to be infinite. In that case a C and as is strings will be output up to but not including a terminating null character and entire Pascal strings will be output.

flags	-	left-justify
	+	*ignored*
	space	*ignored*
	#	the corresponding parameter is treated as an unsigned long, and printed as if it contains 4 characters

P Unless the "#" flag is used, the corresponding arg parameter must be a Pascal string type (or a pointer) and the value is copied to the output as is. When the "#" flag is used, the corresponding parameter is treated as an unsigned long, and printed as if it contains 4 characters.

precision The precision specifies the maximum number of characters to output. The default precision is assumed to be infinite. In that case the entire Pascal string will be output.

flags	-	left-justify
	+	*ignored*
	space	*ignored*
	#	the corresponding parameter is treated as an unsigned long, and printed as if it contains 4 characters

Note: You must use an upper-case %P as shown to output a Pascal string type. If you use a lower-case %p argument, the value displayed will be a pointer type, which is a hexadecimal number optionally preceded by 0X.

t The corresponding arg parameter is converted as a function of its type as follows:

a base type u, d, g, p, or s as appropriate to the type with the precision and flags interpreted as a function of these format codes.

non-base type The value(s) are displayed using a pseudo-Pascal type specification format appropriate to the type of the parameter (e.g. a RECORD/struct type is displayed using a Pascal-like RECORD notation). The flags control some of the aspects of the formatted output.

Note that the corresponding arg parameter need not specify a value and instead may specify only a type. In this case, the type definition is displayed, again using the same pseudo-Pascal type specification format.

flags	-	display only the type even if corresponding arg parameter specifies a value. The type is to be displayed exhaustively, i.e., display every type down to its base type.
	+	display only the type even if corresponding arg parameter specifies a value.
	space	show record/struct field offsets
	#	show all values and offsets in hexadecimal

% A single "%" is output; no arg is used.

precision *ignored*

flags	-	left-justify
	+	*ignored*

space	*ignored*
#	*ignored*

Listing Program Information

The List command can be used to display a list of current processes or local program symbols, as well as to display a list of address and trap breakpoints (or tracepoints). The format of the List command includes four different options: symbol, process, break, or trace. These options are shown below:

list symbol

or

list process

or

list break [{**traps** | **addrs**}]

or

list trace [{**traps** | **addrs**}]

For program symbols, the display includes the local procedure, the parent procedure, the locally-defined variables, any procedures called by the local procedure, and any types defined in the local procedure. Note that in this display, the term "module" denotes a procedure within a program.

For example, to list the local symbols for the DisplayText procedure, use List as follows:

```
Module DisplayText.(0)
  Parent Module
    TransDisplay
  Variables
    theText
    len
  Contained Modules
    None.
  Types
    None.
```

For processes, the display includes the following information: a process number, a "loaded" or "unloaded" designation, and the filename for the process. The process numbers are incremented up to a value of 16 as each new process is started on the system. When you quit from an application, however, its process number isn't reassigned; the next process started will have a new process ID number.

The example below shows a number of processes that were running when Eventlog was suspended. The suspended process is indicated by a bullet (•).

```
  Process#  Loaded?     FileName
•        8  Loaded      "Eventlog"
         5  Loaded      "SADE"
         4  Loaded      "Microsoft Word 3.01"
         2  Loaded      "Finder"
```

For breakpoints and tracepoints, List displays the location and the symbolic representation for the location when sufficient symbolic debug information is present. If the **traps** or **addrs** modifiers are present,the list will be restricted to the specified class of breakpoint. For trap breakpoints, the names of traps (or ranges of traps) with breakpoints set are displayed.

Special-purpose Displays

The SADE memory display commands listed in this section provide special-purpose displays. These include disassembly of instructions, dumping memory, and displaying stack frames.

The Disasm Command

The Disasm command disassembles instructions and displays the offset, address, hex representation, and the assembly code for each instruction. Its format is:

> **disasm** [*addr* [*count*]]
>
> or
>
> **disasm** [*addr range*]

The default behavior when no address is specified is to begin disassembling at the end of the last disassembly. If the value of the program counter has changed since the last disassembly, the program counter (PC) is used as the starting address. You can also specify a particular location to begin disassembly by using *addr* or *addr range*. If no range or count is specified, the number of instructions (not lines) disassembled defaults to 20.

Each line of the disassembly output is divided into four fields or areas. The initial value for DisAsmFormat is "OAXC". This means that the initial order of the fields shown during disassembly is offset, address, hex representation, and finally the assembly code. Their display (both order and presence) is controlled by the DisAsmFormat builtin variable as described in Chapter 3.

For example, to diassemble 5 instructions starting at the eighth statement of the DisplayText routine, enter:

```
 disasm DisplayText.(8) 5
 DisplayText
+0040  003191A8  2F2D FE64    MOVE.L    -$019C(A5),-(A7)
+0044  003191AC  2F2D FE78    MOVE.L    -$0188(A5),-(A7)
+0048  003191B0  4EBA FE26    JSR       FlushDWindow       ; 00318FD8
+004C  003191B4  486D FE48    PEA       -$01B8(A5)
+0050  003191B8  4EBA 00C0    JSR       DisplayString      ; 0031927A
```

Compare this with the output of the Dump command, shown in the next section.

The Dump Command

The Dump command displays a portion of memory at the specified location within a program. Just entering "dump" with no parameters will display the instruction in memory at the current program counter location. You can also display the locations specified by *addr* or *addr range*. The memory is displayed in hexadecimal and ACSII characters according to the specified grouping, which may be byte, word, or long. The default grouping is word.

For example, to display some of the memory area from the DisplayText routine, using the default word grouping, use Dump as shown below:

```
dump DisplayText.(0)..DisplayText.(8)

00319168  4E56 FFF2 48E7 0F18  286E 0008 4AAD FE60  NV..H...(n..J..`
0319178  6700 00E8 486E FFFC  A874 2F2D FE60 A873  g...Hn...t/-.`.s
00319188  2F2D FE60 4EBA F606  266D FE74 2053 3028  /-.`N...&m.t S0(
00319198  003C 48C0 222E 000C  D280 B2AD FE68 6F14  .<H."........ho.
003191A8  2F                                       /
```

The Stack Command

A look at the stack can help you determine what procedures had been called at the time your application was interrupted. To display a list of the stack frames for the current target application, use the Stack command. Its format is:

stack [*count*] [**at** *addr*]

The stack frames displayed are based on register A6 or *addr* if **at** is specified. For each stack frame, the contents of the frame pointer indicated where the frame starts. The frame owner corresponds to the portion of the program that allocated the frame. The procedure that called the frame is listed with an offset if needed.

If an explicit *count* is specified, then at most that many stack frames will be displayed.

For example, the stack frame for the "Eventlog" sample program when it is interrupted at the DisplayText procedure is:

```
launch ":Eventlog"
go til DisplayText

stack
 Frame Addr  Frame Owner   Called From
  <main>      CMain
  $0032BC24   main          CMain+$0028
  $0032BB2C   SkelMain      main.(51)
  $0032BB0C   LogEvent      SkelMain.(13)+$0012
  $0032BADC   ReportUpdate  LogEvent.(50)+$0004
  $0032BACC   DisplayText   ReportUpdate.(1)+$0004
```

This shows that the last stack frame used was from the procedure ReportUpdate, which is part of the "Eventlog" program.

Displaying and Checking the Heap

This section describes the SADE heap commands, which act upon the current application heap zone. The application heap is where the bulk of your program code resides. The heap is divided into a number of blocks, each with a master pointer. There are a number of conditions that can apply to each block on the heap: it may be free or in use, relocatable or nonrelocatable, locked or unlocked, purgeable or nonpurgeable, and may or may not contain resources. Each of these conditions have implications for the application's performance. A complete discussion of the application heap zone can be found in several different books, including *Inside Macintosh.*

The SADE heap commands let you display this wealth of heap information or check the blocks on the heap for consistency. The Heap [display] command can be used to give a complete snapshot of the heap at the time your application was interrupted. Its format is:

> **heap** [**display**] [*addr*] [, *blocktype*]
>
> where
>
> *addr* is an address expression
>
> *blocktype* is a string expression, of which only the first 4 characters are significant. It must be one of the following values:
>
> > 'purgeable' will limit the display to purgeable blocks
> >
> > 'nonreloc[atable]' will limit the display to nonrelocatable blocks.
> >
> > 'reloc[atable]' will limit the display to relocatable blocks
> >
> > 'free' will limit the display to free blocks.
> >
> > 'lock[ed]' will limit the display to locked blocks.
> >
> > 'res[ource] will limit the display to resources
> >
> > 'restype *type*' will limit the display to the specified resource type

If desired, you can display the heap that starts at *addr*. The default is to display the heap pointed at by theZone. By default, the information displayed is:

> the address of the beginning of the heap block
>
> the address of the master pointer if it's a relocatable block
>
> an asterisk if the object is locked or nonrelocatable,
>
> the value of the tag byte (for relocatables)
>
> for a resource, the reference number of the file it's in, and the resource type and ID of the resource

To display a subset of the heap objects, you can specify one of the block types. The *blocktype* must be one of the following values: purgeable, non-relocatable, relocatable, free, locked, resource, or a particular resource type.

Using the "Eventlog" example, only resources of type 'MENU' are displayed when the Heap display command is used as follows:

```
Heap display restype 'MENU'
        BlkAddr     BlkLength Typ MasterPtr  Flags RType   RId RFRef  RName
        $00316590  $00000098  H  $0031452C    R    MENU   1000 $0584  "File"
        $00316838  $00000050  H  $00314528    R    MENU   1001 $0584  "Edit"
        $00316888  $000000F4  H  $00314524    R    MENU   1002 $0584  "Log"
```

The Heap totals command provides a way to display summary information about the heap. Its format is:

> **heap totals** [*addr*] [, *blocktype*]
>
> where
>
> *addr* is an address expression
>
> *blocktype* is a string expression, of which only the first 4 characters are significant.

The type information follows the same rules as the Heap command. The summary shown is for the heap that starts at *addr* for blocks of type *blocktype* . The summary information is given for free, nonrelocatable, and relocatable objects in the heap unless *blocktype* is specified. If *blocktype* is specified, the summary information is limited to the indicated type of object.

For instance, the summary information for purgeable blocks within the "Eventlog" sample program provides the following display:

```
heap totals purgeable

                    Total Blks  Total Size
Purgeable                    8        8816
```

To check the consistency of the heap for the current target program, use the Heap check command. Its format is:

> **heap check** [*addr*]

The Heap check command checks the consistency of the heap . If desired, you can check only that part of the heap that starts at *addr*. The default is to display the heap pointed at by theZone.

The Heap check command performs range checking to make sure all pointers are even and non-NIL, and that block sizes are within the range of the heap. It then makes sure that the self-relative handle points to a master pointer referring to the same block. For non-relocatable blocks, it checks if the heap zone pointer points to the zone where the block exists. The command also verifies that the total amount of free space is equal to the amount specified in the heap zone header, that all pointers in the free master pointer list are in master pointer blocks, and does other header validation.

Displaying and Checking Resources

No Macintosh debugging system would be complete without a way to display resources. In SADE, the Resource command is used to display and check the resource map for an application. This command has two formats: Resource display and Resource check. The format for displaying resource maps is:

resource [**display**] [*addr*] [**restype** 'type']

If an address expression is not specified in *addr* , the default is to display all resource maps for the target application. The information displayed for each resource map includes: its location, the refnum of the resource file, and a list of the instances of each type. For each resource type displayed within the map, the following information is displayed: the resource ID, the resource type, the value of the master pointer, whether the resource is locked or unlocked, and the resource name.

The following example from the "Eventlog" program shows a partial resource map, using only the resources of restype 'WIND'.

```
resource restype 'WIND'
Resource Map at $00316EF8
   ResId  RType  MasterPtr  Locked?   Name
    1000  WIND   $00316BE4  Unlocked
    1001  WIND   $00316C08  Unlocked
    1002  WIND   $00316484  Unlocked
    1003  WIND   $003164B8  Unlocked
    1004  WIND   $003164D8  Unlocked
Resource Map at $0002B19C
   ResId  RType  MasterPtr  Locked?   Name
  -16000  WIND   NotLoaded
  -15968  WIND   NotLoaded
  -15840  WIND   NotLoaded
```

You can check the resource map for consistency using the Resource check command. Its syntax is:

resource check [*addr*]

If an address expression is not specified in *addr* , the default is to validate all resource maps for the application. If an inconsistency is found, the command displays a diagnostic message specifying the problem.

Finding Program Locations

To find an expression within a program, use the Find command. It evaluates the supplied numeric or string expression, searches for the target, and displays the result. Its syntax is:

find [**count**] *target*[,*n*] [*addr range* [*mask*]]
or
find [**count**] *target*[,*n*] [*addr* [*count*] [*mask*]]

The Find command normally counts or searches all of a program's available code for occurrences of the target. As options, you may start at the specified address and look up to *count* bytes beyond, or limit the search to *addr range*. The default range is HeapStart to MemTop.

The *mask* parameter is an optional numeric or string expression that is logically ANDed to the contents of each memory location before the comparison is done. The *n* parameter is an integer expression that specifies the minimum number of occurrences to ignore. In other words, the first *n*–1 occurrences may be ignored in the search. The default for *n* is 1; in this case, SADE will find the first occurrence. For example, by specifying 3 for *n*, you could find the third occurrence

For example, to find the string "mystring" in your program, enter:

```
find 'mystring'
```

Setting Memory Values

After you have found and displayed a particular program location, you may want to modify its contents. The typical modifications a programmer may make are:

- Modifying a program variable
- Changing a procedure parameter

However, any time you change a program location, you are altering your program and could run into unexpected trouble when you resume execution. It's always a good idea to save the value of whatever variable or parameter you want to change in a debugger variable.

For example, you can change the program variable named "len" (type long) to have a value of 10,000 with an assignment as shown below. First, save the current value of "len" in a debugger variable named saver. Then do the assignment.

```
saver := len
len := 10000
```

Chapter 5
Program Control

This chapter provides an introduction to the SADE's program control functions. After you have launched your program, you can always interrupt it by pushing the NMI key. However, this method is often unsatisfactory, since you normally have no way of knowing exactly where you were in the program at the time you pushed the key. The OnEntry message usually displays the name of a procedure, but you can't tell if that was the third or the thirty-third time the procedure was called. Since these kinds of details are important in debugging, you need to be able to interrupt your program at one or several chosen places in the code, and examine the results.

The SADE program control commands include the following features:

- The Go command resumes program execution, with the option of using a temporary breakpoint or a conditional statement to halt the program later.
- The Step command executes the program one instruction or one line at a time.
- The Break command sets breakpoints on one or more addresses, on trap ranges, or on all traps. A break action may be associated with a breakpoint; this command or group of commands will be executed when the breakpoint is reached.
- The Stop and Abort commands can be used to stop the debugger during break actions.
- The Trace command monitors program execution by writing a message to standard output each time a specified trap or address is reached. Program execution continues during a trace operation.
- The Unbreak and Untrace commands can be used to clear breakpoints and tracepoints.

These program control commands are described in the following sections. The sample program "Eventlog" is used throughout this chapter to show the effect of these commands on a program.

If you have enabled source level debugging on your system through use of the StandardEntry proc in the SADEStartup file, you can also perform some program control operations by selecting items from the SourceCmds menu. See Chapter 8 for more information on source level debugging.

Resume Program Execution

If you've been following the sample program used in previous chapters, you've already seen one way to resume program execution. The Go [til *addr*], form of the Go command is extremely useful for executing your program up to the point you want to examine, then halting it. The breakpoint set with Go [til *addr*] is temporary.

Of course, you don't always know exactly what program location you need to examine. You can make your program resume execution at the current program counter by entering the Go command as follows:

```
go
```

After using the Go command like this, the program will keep running until something interrupts it. The "something" may be a bug, or it may be a breakpoint or tracepoint that you have set, using the other commands in this chapter.

The complete syntax for the Go command is:

> **go** [**til** *addr*], ..
>
> or
>
> **go** [**while** *expr*]
>
> or
>
> **go** [**until** *expr*]

The Go command resumes execution of the target program until a breakpoint is reached, a **while** condition becomes false, or an **until** condition becomes true. At this point, the debugger is entered and control returns to the user.

When you use the Go [til *addr*] format, the debugger sets temporary breakpoints at any addresses you specified. When the breakpoint is encountered, the debugger is reentered and the breakpoint is removed. If the address is in ROM, the debugger will warn you that it can't set a temporary breakpoint in ROM. To set a trap breakpoint, use the Break command as described later in this chapter.

Similarly, if a condition expression is specified, the debugger will run the application in trace mode until the **until** condition is met or or the **while** condition is broken (whichever is appropriate). The choice of **while** or **until** allows you to reverse the sense of the test, but it doesn't affect when the test occurs. During trace mode, program execution continues, but SADE checks to see if the conditional expression is satisfied.

Stepping Through a Program

When you step through a program, you are able to examine the effect of each statement. SADE gives you the option of stepping through one instruction at a time, or stepping through one line of source at a time. The Step command format is:

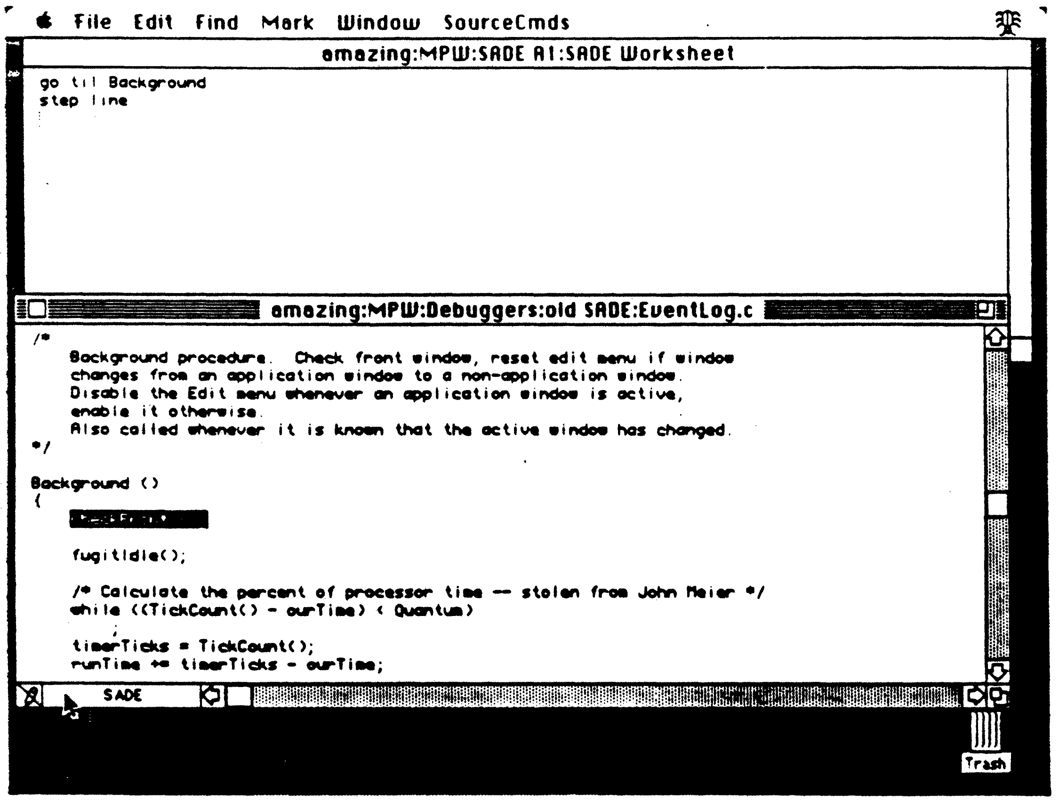
> **step** [{**asm**|**<u>line</u>**}] [**into**]

If **line** (the default) is specified, the debugger will execute all of the instructions associated with the current source line. If **into** is used with the **line** option, each call to a subroutine will cause the debugger to be reentered at the first line of the called routine. Otherwise, routines called by JSRs and BSRs will be treated as single instructions. Trap calls are always stepped over.

If **asm** is specified, the debugger executes the instruction at the current PC location. The debugger is reentered after the instruction executes. If **into** is specified, calling subroutines will cause the debugger to be reentered at the first instruction of the called routine. Otherwise, routines called by JSRs and BSRs will be treated as single instructions.

If you have source level debugging enabled (using the StandardEntry proc in the SADEStartup file), you may display the source window during a step operation. The window containing the current line is displayed, with the next line to be executed highlighted. See Figure 5-1 for an example of this display.

In addition, the Step command can be executed from a selected point within the source file, using the SourceCmds menu item Step. More information on source level debugging is contained in Chapter 8.


File
Edit
Find
Mark
Window
SourceCmds
amazing:MPW:SADE A1:SADE Worksheet
amazing:MPW:Debuggers:old SADE:EventLog.c
SADE
Trash

```
go til Background
step line
```

```
/*
    Background procedure.  Check front window, reset edit menu if window
    changes from an application window to a non-application window.
    Disable the Edit menu whenever an application window is active,
    enable it otherwise.
    Also called whenever it is known that the active window has changed.
*/

Background ()
{
    [illegible]

    fugitIdle();

    /* Calculate the percent of processor time -- stolen from John Meier */
    while ((TickCount() - ourTime) < Quantum)

    timerTicks = TickCount();
    runTime += timerTicks - ourTime;
```

Figure 5-1.
Source Window after Step Command

Suspending Program Execution

SADE allows you to set a breakpoint in one or more program locations, which can be referenced as an expression or an address. The two types of breakpoints used in SADE are known as address breakpoints and trap breakpoints. Both kinds of breakpoints are set using the Break command, and may be optionally be followed by a break action. The breakpoint types and break actions are described in the following sections.

After the desired breakpoints are set, you can then resume program execution. In the simplest cases, the program will run until it reaches the first available breakpoint, and then the debugger will stop program execution. At this point control normally passes back to SADE.

SADE breakpoints and break actions can interact in many different ways with other SADE commands. It's difficult to make a general statement that applies to all uses of breakpoints and break actions. The simplest kind of breakpoint to use is an address breakpoint with no break action. The most complex situations occur when a number of trap breakpoints are set, each having a break action. Break actions themselves may be either simple or may consist of a series of commands grouped with Begin..end.

The commands within a program also have an effect on SADE's breakpoint interpretation. If the last program execution command was in a structured statement, and no break action was specified for the breakpoint, the commands immediately following the program execution command are also executed.

Address Breakpoints

An address breakpoint can be set anywhere within program code by specifying a RAM address. Address breakpoints may also be set using symbolic references; in this case the code may not yet be in memory at the time the breakpoint is set. The form of the Break command used for address breakpoints is:

> **break** *addr* **, ...** [*break action*]

A *break action* may be a single debugger command, a debugger proc call, or group of commands delimited by Begin..end. Break actions, and their interaction with other SADE commands, are described below.

Trap Breakpoints

Trap breakpoints may be set on a range of traps, or on all traps. For trap ranges, either the trap name or the trap number can be used in a range expression. Trap numbers are prefixed with a "†". The two forms of the Break command used to set trap breakpoints are:

> **break** *trap range* [**from** *addr range*] [*break action*]

or

break all traps [**from** *addr range*] [*break action*]

The *break action* may a single debugger command, a debugger proc call, or group of commands delimited by Begin..end, as described below. The *trap range* is a address range beginning and ending with trap names of the form †$Axxx..†$Axxx.

The same trap may have a breakpoint set on it by its name alone, as a member of a trap range, or as a trap within a specified address range. Each of these trap breakpoints will have a separate breakpoint record created for it; you can see what breakpoints are set using the List break command.

The same trap can be specified in multiple break commands. This can happen in one of two ways: either overlapping trap ranges are specified, or different address ranges are specified. Consider the following examples where multiple break commands are set with overlapping ranges:

NOTE: These examples have not been tested and may not work on your program!

```
break all traps action1          # break on any trap called from any address

break †$A996..†$A9A0 action2             # break on _OpenResFile (and other traps)
                                         # called from any address

Break _OpenResFile from applZone..applZone^ action3     #break on
                                 # _OpenResFile only if called from application heap

break _OpenResFile from ROMBase..ROMBase+256K action4
                                 # break on _OpenResFile only if  called from ROM
```

With the above trap breakpoints set, the program's calls to _OpenResFile would be handled as follows:

address	call	action
$10000	_OpenResFile	action 2 is interpreted. SADE first looked for breaks on _OpenResFile only, but the address fell outside of the specified ranges. It then found the first (most recently specified) trap range that included _OpenResFile.
myproc+$100	_OpenResFile	action3 is interpreted. This call occurred within the application heap

SADE uses a search rule when finding an instance of a trap call that satisfies the conditions for a specified breakpoint. For any breakpoint that specifies a trap name, SADE first attempts to find a call within the program that exactly matches that name. Once it finds the trap, SADE checks to see if it falls into the address range if one was specified.

The next step is to check any trap ranges that might contain the trap. SADE accepts the most recently defined trap range it finds that contains the trap, even if another trap range defined earlier also contains the trap. Once a trap range satisfying the description is found, SADE then checks to see if the range falls within the address range if one was specified.

The Break all traps option sets a trap breakpoint on all traps in a contingous range. If *addr range* is specified, the debugger will break on traps called from the specified memory range.

Break Actions

A break action is one or more SADE commands that are meant to be interpreted after the breakpoint is reached. Normally, once SADE has performed the commands within the break action, program execution will continue. However, using a Stop or Abort command as part of the break action may halt subsequent program execution.

If a Stop command is contained in a break action, and the most recent program execution command was in a structured statement, the commands immediately following the program execution command are also executed.

If an Abort command is encountered in a break action, all pending program execution commands are aborted. The debugger is entered immediately.

Note that the commands specified in the breakpoint action are saved and not interpreted until the time when the breakpoint is reached. Consequently, any SADE variable references used in a break action should use global variables only. Local variables defined in a break action won't exist at the point when the breakpoint is reached and the break action is invoked.

Unbreak Command

To clear a break on traps within the specified addresses or range of traps, use the Unbreak command. This will also remove their associated break actions. The command format is:

> **unbreak** *addr* , ...
>
> *or*
> **unbreak** *trap range* , ...
>
> *or*
> **unbreak all** [{**traps** | **addrs**}]

The *addr* is an address expression, and the *trap range* is a range of trap locations. The **all** form will clear all breaks, optionally restricted to just traps or addresses if the **traps** or **addrs** modifiers are present.

For example, to undo a break on the GetResource trap, enter:

```
unbreak _GetResource
```

Stop Command

To return control to SADE while a program is executing, use the Stop command. Its syntax is simply:

stop

The Stop command terminates any debugger commands already in progress. This command can be used to stop the debugger during the execution of break actions. See the section on "Break Actions" for more information about the use of Stop.

Abort Command

The Abort command terminates the current break action, causes the debugger to be fully entered, and cancels any debugger commands pending. This means that when the previous execution was in a structured statement, the pending commands are canceled. The command syntax is simply:

abort

See the section on "Break Actions" for more information about the use of Abort. See also the Stop command, which terminates the break action without cancelling other pending commands.

Monitoring Program Execution

Tracepoints allow you to monitor addresses or traps during program execution, without halting the program. When the tracepoint is encountered, a message is written to standard output reporting the address or trap being traced, or the symbolic representation of the address if available. This section describes the commands for setting and undoing tracepoints.

Trace Command

The Trace command sets tracepoints on the specified address or traps within the target application. After setting the tracepoints, you can resume program execution. When the tracepoint is encountered in the executing program, a message is displayed on the current standard output, reporting the address or trap being traced, and optionally the symbolic representation of the address. The command format is:

trace *addr* ,...

or
trace *trap range* [**from** *addr range*] , ..

or
trace all traps [**from** *addr range*]

If *addr range* is specified, the message will be written only if the trap was called from the specified memory range. In any case, execution is resumed after the message is displayed.

For trap ranges, either the trap name or the trap number can be used in a range expression. Trap numbers may be prefixed with a "†"; for example, the range from the system trap _OpenResFile to _GetResource could be specified as

```
†$A997..†$A9A0
```

For example, to use a trap range with the Trace command, you can enter:

```
trace _OpenResFile.._GetResource
```

Untrace Command

The Untrace command clears the tracepoint for the specified addresses or traps. Its format is:

untrace *addr* ,...

or
untrace *trap range* ,

or
untrace all [{**traps** | **addrs**}]

The *addr* is an address expression, and the *trap range* is a range of trap locations. The **all** form will clear all tracepoints, optionally restricted to just traps or addresses if the **traps** or **addrs** modifiers are present.

For example, to undo a trace on the _GetResource trap, enter:

```
untrace _GetResource
```

Chapter 6

Debugger Command Flow Control

This chapter provides an introduction to the SADE execution flow control commands. These commands are used in conjunction with other SADE commands to control the sequence of debugger execution. Flow control commands are useful for automating your debugging session: you can execute a group of commands as one unit, specify alternate actions depending on the values within your program, or repeat debugger actions for a specified number of times. With flow control, the tests you set for your program code will only be performed under the conditions you specify.

The flow control commands can be divided into three groups:

- Grouping commands, which include the Begin...end construct.
- Conditional execution commands, including the If...end construct, with the optional Else and Elseif commands.
- Looping commands, including the For, While, Repeat, Loop, Cycle, and Leave commands.

The syntax of these commands is similar to that of the structured statements found in high-level programming languages: a Begin must be followed by an End, a For is used with a control variable, and an If can have alternate conditions using an Elseif or Else. You can find many examples of flow control commands in the command files in the SADEScripts folder.

Grouping Commands

The grouping commands, Begin...end, allow a series of commands to be interpreted by SADE as a single unit. All of the commands within the Begin..end construct are evaluated before SADE executes any of them. The component commands are executed in the same sequence as they are written.. The command format is:

begin

commands # all commands are evaluated before execution

end

The SADE Worksheet includes an example of the difference between commmands executed separately and those that are executed as a group.

```
# with a target suspended, select and execute the following statements
begin
	step asm over
	printf "this statement will execute when Sade is reentered\n"
end
```

```
# note the different behavior if the following statements are executed
# instead

step asm over
printf "this statement will not be executed when Sade is reentered\n"

# this difference may change in the next release
```

One use of the Begin..end construct is the specification of breakpoint actions which consist of more than one command or procedure invocation. If you don't use a Begin...end grouping, you can only specify one command or one procedure invocation after your breakpoint. For instance, the following example is a breakpoint set on the procedure DisplayText. The StandardEntry proc is the break action when SADE is entered.

```
break DisplayText StandardEntry     # use proc from the SADEStartup file
```

However, when you use Begin..end, you can use any number of commands following a breakpoint:

```
break DisplayText begin
      printf "watch out for bugs"   # print a message to yourself
      list symbol             # list local symbols for this procedure
      resource restype 'WIND' # display 'WIND' resources
      end
```

The Stop and Abort commands, described in the previous chapter, may be used within a Begin..end construct to halt execution of the break action. The example below shows a Begin..end construct used with the If and Stop commands. If the condition specified in the If statement is true, the break action will halt execution.

```
break DisplayString.(4) begin
      str := theStr^            # save value of parameter in debug variable str
      if str = '***' then stop   # halt break action if this condition is true
      printf "%t\n", str         # otherwise continue break action execution
      end
```

Conditional Commands

Conditional execution of debugger commands allows you to specify a number of alternate actions, each based upon the the evaluation of a boolean expression. If the boolean expression produces the value true, the commands that follow can then be executed. The If..end command format is:

> **if** *booleanexpr* [**then**]
>   *command*
> [**elseif** *booleanexpr* [**then**]
>   *commands*..]*...
> [**else**
>   *commands*...]...
> **end**

Each If command must be followed by an End command. Elseif and Else commands are optional, but must appear between the If..end commands in the order indicated above. More than one Elseif may appear (indicated above by the [...]*); but at most one Else may appear.

When an If ... end construct is evaluated, each of the statements is checked in order of appearance. When the first If condition (a boolean expression) is true, the statements controlled by the If are executed and the remainder of the contruct to the End is skipped. If the condition is false, the statements controlled by the If are skipped and the next (Elseif) condition is checked, if present. If an Elseif condition is evaluated and is true, the commands it controls are executed and the remainder of the If construct is skipped. If no previous conditions were evaluated as true when the Else command is reached (if present), the commands controlled by the Else are then executed; otherwise they are skipped.

The commands controlled by an If extend to the corresponding End, or to the first Elseif or Else, whichever comes first. The commands controlled by an Elseif extend to the next Elseif, Else or End, whichever comes first. The commands controlled by an Else extend to the corresponding End.

If...end constructs may be nested. They may also be used in combination with the other flow control commands, as seen below. This example is from the file DisplayMemory in the SADEScripts folder.

```
proc dm a,n          # display memory from address a for n bytes
   define i, j, b, s := ''                        # start with nulls
   If undef(n) then
        n := 16                                   # [default n = 16]
   end
   for i := 1 to n do                             # outer for loop
        if (i mod 16) = 1 then                    # outer if statement
             if s <> '' then                      # 1st nested if
                  printf " '%t'", s
             end
             if i <> 1 then                       # 2nd nested if
                  printf "\n"
             end
             printf "%.8X: ", a
             s := ''
        end                                       # end of outer if
        b := ^unsignedByte(a)^
        printf "%.2X ", b
        if (b < $20) | (b > $7E) then             # if..else construct
             s := concat(s, '.')
        else
             s := concat(s, cChar(b))
        end
        a := a + 1
   end                                            # end of for loop
   if s <> '' then                                # outer if statement
     if i > 16 then                               # 1st nested if
             i := i mod 16
             if i <> 0 then                       # 2nd nested if
                  for j := i to 15 do             # nested for loop
                       printf "   "
                  end
             end                                  # end of 2nd nested if
        end                                       # end of 1st nested if
        printf " '%t'", s
   end                                            # end of outer if
   printf "\n"
end                                               # end of proc
```

To use this proc, supply a memory location and the number of bytes to be displayed:

```
dm $319168,16
00319168: 4E 56 FF F2 48 E7 0F 18 28 6E 00 08 4A AD FE 60  'NV..H...(n..J..`'
```

This performs the same operation as:

```
    dump byte $319168
```

Looping Commands

This section describes SADE's looping commands, which allow you to execute one or more SADE commands repeatedly. You can loop for a specified number of times with For, loop unconditionally with Loop, or loop with a test at the beginning(While) or the end (Repeat). The Cycle command allows you to execute within a loop construct.

For Command

The For command allows you to repeat a one or more SADE commands repeatedly, using a control variable to specify the number of repetitions. The command format is:

> **for** *for clause* [**do**]
>
> *commands*
>
> **end**

The commands enclosed in the For..end construct are executed until the control variable has taken on each successive value in the range expressed by the *for clause*. The *for clause* is composed of debugger variables and expressions. It may have one of the following forms:

var = *expr* **to** *expr*	The first expression is the initial value, and the second expression is the final value. The commands are executed once for every value in this range.
var := *expr* **downto** *expr*	With **downto**, the value of the control variable is decremented by one for each repetition, starting with the initial value and conclusing at the final value.
var := *expr* , ...	A list of expressions may be used. Execution continues until the control variable has taken the value of each of the listed expressions.

See Chapter 3 for more information on the proper format for debugger variables.

For example, to perform SADE commands starting with a control variable of x equal to 10 until the control variable equals 5, use:

```
for x := 10 downto 5 do
   (commands)      # these commands will execute 5 times
end
```

For..end commands may be nested. They may also be used with other flow control constructs, as well as in break actions.

While Command

A While statement contains an boolean expression at the beginning of the construct, which is evaluated before execution takes place. The commands contained in the While...end construct will only execute when the expression is true. If the expression is false, the enclosed commands are skipped, and execution resumes following the end. The command format is:

```
while boolean expr [ do ]
   commands
end
```

While..end constructs may be nested.

The following example is from SADEScripts:MiscProcs. It uses a While..end construct to check each window in the window list, and stops when the NIL pointer at the end of the window list is encountered.

```
proc displaywindowlist;
# displays the address and title of each window in the window list.

define nextwindow;             #used to contain the pointer to each
                               # window record in turn
define wTitleoffset := $86;    #from ToolEqu.a.  the offset into the
                               # window record to the handle to the
                               # window title.
define wTitle;                 #used internally to point to the window
                               # title.

nextwindow := windowlist;      # start at the first window, pointed to
                               # from low memory.

while (nextwindow <> 0) do     # a NIL terminated list.
      wTitle := (nextwindow+wTitleoffset);
                               # point at the window's title
      printf ("nextwindow: $%.8X Window Title = ∂
      \"%p\"\n",nextwindow,^pstring(wTitle^^^)^);
                               # write the information out.

      nextwindow := (nextwindow+$90)^;
                               # point at the next in the list.
end; # while

end; # displaywindowlist
```

Repeat Command

To repeat a group of commands with a conditional test at the end of the loop, use the Repeat command. The SADE commands enclosed by the Repeat..until construct are executed until *boolean expr* is evaluated as true. This ensures that the enclosed commands are executed at least once.Repeat constructs may be nested.

The command format is:

repeat

commands

until *boolean expr*

For example, to use a Repeat..until construct that tests the value of variable x:

```
repeat
   <supply commands here>
until x = 5
```

The SADEScripts:ResVerify command file contains a Repeat..until construct that executes once for each resource map. It stops execution when there are no more resource maps to check. A portion of the ResVerify proc appears below:

```
proc ResVerify
      # verifies all resource maps in the resource chain
      # and all loaded resources

<definitions for local variables>

      repeat

            NEXTresFile := nextResHndl^
            < commands here to check resource maps and display results>
            nextResHndl := (NEXTresFile + 16)^
            # check next resource map

      until (nextResHndl = 0) # until no more resource maps to check
end   # proc ResVerify
```

Loop and Leave Commands

The Loop..end construct provides unconditional looping. Its format is:

loop

commands

end

The enclosed SADE commands are executed repeatedly. Loop..end constructs may be nested.

To exit the loop use the Leave command. Its format is simply:

leave

For example, you could use Loop and Leave as shown below:

```
loop
   < execute commands here>
   leave if i = 5
   < otherwise execute more commands>
end
```

Cycle Command

The Cycle command can be used with any of the While, Repeat, Loop, or For constructs. It causes execution to continue from the top of the currently enclosing construct. Its format is:

cycle [**if** *boolean expr*]

If the optional If clause is present, the cycle action will happen only if the *boolean expr* is true; otherwise execution will continue following the Cycle command.To continue execution within one of the other looping constucts, use the Cycle command.

For example, to use Cycle within a For construct:

```
for  x := 10 downto 5 do
   <commands>
   cycle if x < 2
   <more commands>      # executed only when expression is false
end
```

Chapter 7

Customizing the Debugging Environment

One of the most powerful aspects of SADE is the extent to which you can customize your debugging sessions. In earlier chapters, you've seen how SADE commands include many options that allow you to display information in the format you specify, or to execute debugger commands in the sequence you choose. This chapter describes some of the ways you can customize the SADE user interface—the menus that appear, the messages that are displayed when you enter the debugger, and the functions and procedures that perform specialized tasks.

You can extend SADE's capabilities in one or all of the areas listed below:

- Create a SADE file, fill it with a series of SADE commands, and execute those commands by invoking the command file name.
- Define a function or procedure in the SADE command language. These functions and procedures can be called from within a SADE command file. Function and procedure names, and any variables defined within them, are interpreted according to the scope guidelines in Chapter 3.
- Create abbreviations for command names with user-defined macros. You can choose an identifier that will stand for for any string expression you specify.
- Customize the SADEStartup file to include the initialization information you want when you launch your SADE debugging session.
- Supply OnEntry actions that will be executed each time the Debugger is entered.
- Add menus and menu items, allowing you to select which debugger commands are available from the SADE menu bar. You can delete those menu items which aren't useful to you.
- Add alert boxes to inform you when selected debugger operations have occurred

The SADEScripts folder contains a number of SADE command files; these will give you an idea of some of the ways that SADE commands can work together to provide a customized debugging environment. As you read this chapter, you'll see several examples of SADE procs, funcs, and macros; these examples can also be found in the SADE Worksheet and the SADEStartup file.

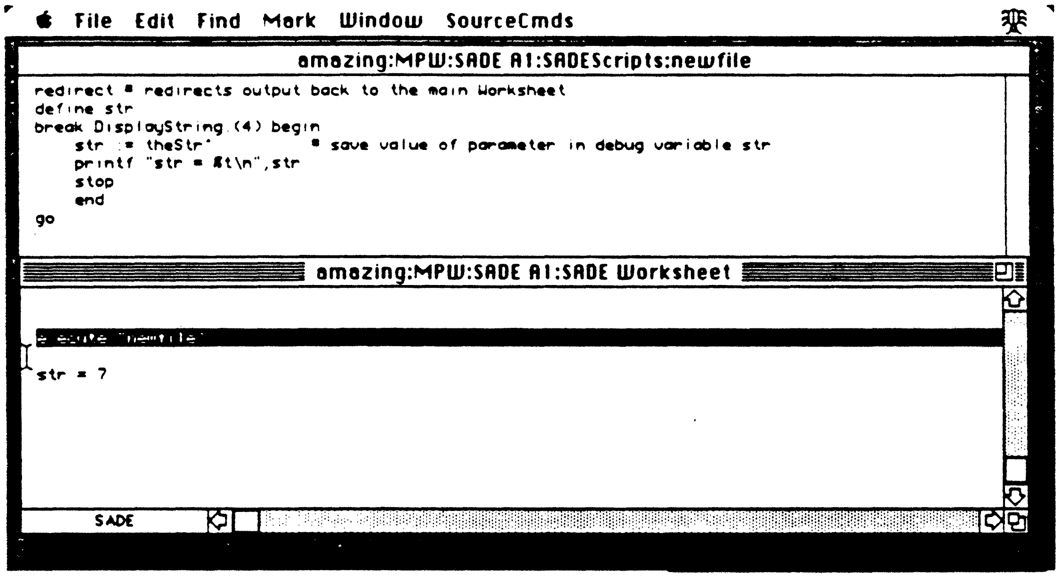
Executing a Debugger Command File

If a file or window contains executable commands, the Execute command can be used with the filename to execute those commands. Any file that contains a group of SADE commands can be used as a command file. The commands within the file can be executed simply by using the Execute command with the filename as a parameter. The syntax of the Execute command is:

> **execute** *filename*

The *filename* is a string expression. The Execute command can't be used in any structured statement, such as a Begin..end construct.

If you have a set of commands in your worksheet that you wish to save for later use, you can open a new SADE file, and place the commands you want to save into the new file. Later, you can use the Execute command to repeat the operations you saved. See Figure 7-1 for an example.


File Edit Find Mark Window SourceCmds
amazing:MPW:SADE A1:SADEScripts:newfile
amazing:MPW:SADE A1:SADE Worksheet
SADE

```
redirect • redirects output back to the main Worksheet
define str
break DisplayString.(4) begin
    str := theStr^           • save value of parameter in debug variable str
    printf "str = %t\n",str
    stop
    end
go
```

```
execute "newfile"

str = 7
```

Figure 7-1.
Creating a new command file for execution

User-defined Macros

The Macro command allows you to define a macro, which associates a string of characters with a named identifier. When SADE encounters these characters, it interprets the macro as if it was dealing with the identifier. For instance, when the macro is used in place of a command, SADE temporarily redirects input to the command that forms the macro's actual value.

The command syntax is:

macro *name string expr*

References to a macro may appear anywhere a token could appear in the debugger input stream. Macro definitions are limited to a length of 254 characters. Macros allow you to create abbreviations, which could be used to change the debugger language to a more terse, Macsbug-like language. (See the example below, which is from the SADEStartup file.)

```
macro br   'break'
macro clr  'unbreak all'
```

To suppress a macro call, you can use an identifier preceded by a "∂". Macro definitions can be nested; that is, a macro definition can contain references to other macros. Macro definitions are not recursive; references in a macro definition to the macro being expanded aren't treated as macro calls. A macro definition is not allowed in any structured statement.

Macros may be redefined. You can remove a macro definition with the Undefine command. This is occasionally necessary if your program contains a variable or procedure name identical to one of the macros defined in the SADEStartup file.

User-defined procedures

Procedure definitions in the debugger language are delimited by the Proc and end commands. The Proc command identifies the procedure's name and optional parameter list. Its format is:

> **proc** *name* [*arg name* , ...]
>   *body*
> **end**
> *or*
> **proc** *name* ([*arg name* , ...])
>   *body*
> **end**

The *name* is a string expression, the *arg name* is a SADE identifier, and the *body* is composed of one or more SADE commands.

A debugger proc is called by beginning a debugger command with the name of the proc, followed by the optional actual parameter list. The parameter list follows the prototype parameters from the proc's definition, with the actual parameter values substituted for the parameter names. When the proc executes, the actual parameter values will be matched positionally with the formal parameter names.

The number of actual parameters need not match the number of formal parameters in the proc definition. If too few actual parameters are specified, the formal parameters for which there were no corresponding actual parameters will be assigned a special undefined value. Extra actual parameters have no corresponding formal name, but can be referenced through the predefined debugger array variable Arg (See Chapter 3). This allows the parameters of a proc to be accessed positionally with references of the form "arg[n]". The number of the last actual parameter specified is contained in the predefined debugger variable NArgs. The values of these predefined debugger variables represent the parameter state of the currently active proc and are not defined when debugger execution is not in a proc.

If a parameter list is used with the Proc command, it identifies parameters by name only. Parameters are not typed within Proc definitions; instead, they take on the types of their actual parameter values at the point of call.

The parameter list may optionally be enclosed in parentheses. If the parentheses are included in the proc definition, they must be included when the proc is called as well. Similarly, if the definition was not parenthesized, then invocations of the proc must not be parenthesized.

Procs may be redefined. A proc must have been defined before a call to it is processed (so that the proc name can be recognized as a debugger symbol). Thus, if mutually recursive procs are desired, one proc must be defined first with a dummy proc definition, so that the second proc can refer to it, and then the first proc can be redefined, referencing the second proc. The minimal dummy proc definition is: "proc foo; end;". Proc definitions may also be removed with the Undefine command.

Proc calls may be nested. The Return command may be used with Proc definitions; see the section which follows for more information.

The following example is from the SADE Worksheet:

```
proc factorial n, file          # the proc Factorial is called with
                                # two arguments
define i
if nargs > 1 then
   redirect file
   end
for i := 1 to n do
   printf("fact(%.2d) = %19.19g\n", i, fact(i))
                          # this line calls the SADE function fact
   end
if nargs > 1 then
   redirect
   end
end   # end of proc Factorial
```

User-defined Functions

Function definitions in the SADE language are delimited by the Func..end construct. Aside from these delimiters, procedure and function definitions are essentially the same. The format for a function definition is:

func *name* ([*param name* , ...])

body

end

The *name* is the function name, the *param name* is the parameter name, and the *body* is the code that makes up the function. A function definition must have a Return command, with a return value specified, as the last executable statement. See the syntax for the Return command below.

The function type isn't specified in the function definition. The function takes on the type of the value specified in the return statement that was executed to leave the function . This means that functions are not limited to returning results of a single type.

A debugger func is called using conventional functional notation, with the function name followed by the optional actual parameters in a parenthesized list in the format of the prototype established in the func definition. Func parameters are handled in the same fashion as Proc parameters, and the predefined debugger variables Arg and NArgs may also be used.(See Chapter 3). User-defined functions may be called anywhere an *expr* is allowed.

Function names can be removed as SADE symbols through use of the Undefine command.

The following example of a function definition is found in the SADE Worksheet:

```
func fact(n)                # supply one argument
  if n <= 1.0 then
     return 1.0
  else
     return n * fact(n-1)   # return the result
  end
end
```

The Return command causes the debugger to exit the a debugger procedure or function currently in execution. If returning from a function, an expression must be specified for the function value. When returning from procedures there should be no return value.

The command format is:

return [*expr*]

Customizing your Startup File

The commands within the SADEStartup file are executed every time you launch the SADE program. You can, if you wish, edit this file, and change the contents to suit your debugging needs.There are a number of different ways that you can customize the SADEStartup file. You can, of course, define your own SADE procs, funcs, and macros, as described above, and place these definitions in the startup file. Or you may place the definitions in command files, and execute the various command files from SADEStartup.

Alternatively, you can leave SADEStartup as it is, and create your own Startup file. For instance, you could name your startup file "UserStartup", and place your procs, funcs, and macros there. To execute "UserStartup" each time SADE is launched, use the Execute command as follows:

```
execute 'UserStartup'
```

You also have the option of changing the default source file display interface that is established in SADEStartup. The commands used to implement the source file display are described in the sections that follow. These include the OnEntry command, which controls SADE's behavior each time the debugger is entered. The Addmenu and DeleteMenu commands let you customize SADE's menu interface. The Alert command gives you the ability to display an alert box whenever you think you need it. Using one or more of these commands, you can substantially affect how SADE will work for you.

The OnEntry Command

The OnEntry command can be used to execute one simple or compound SADE command each time SADE is entered. It is typically used within a startup file to provide a source display each time SADE is entered. Break actions specified using OnEntry may also be used when entering SADE from a breakpoint. The command syntax is:

onEntry [break action]

The *break action* may be one simple SADE command, as shown below:

```
onEntry printf "%.8X\n",pc
```

If more than one command or procedure invocation is needed within the break action, the Begin..end constuct should be used to group a sequence of commands. A compound command formed in this way is interpreted by SADE as one unit.

In the SADEStartup file, the "StandardEntry" proc is the default argument to the OnEntry command. This procedure provides for source display of the current program counter (PC) on entry into the debugger. If you wish to change the behavior of the source display when the debugger is entered, you can write your own SADE procedure, and use its name as the argument to OnEntry.

User-Defined Menus and Alerts

Menu items may be added to, or deleted from, the standard SADE menus File, Edit, Find, Mark, and Window. Within the SADEStartup file, the SourceCmds menu is implemented with a list of Addmenu commands. You may wish to change the contents of the

SourceCmds menu, or to create your own menu containing a different set of debugger commands. The Addmenu and Deletement commands are described below.

The default source display interface uses an alert box to inform you when it has reached a source break. This alert box is implemented with an Alert command in the SADEStartup file. If you wish to change this behavior, or to call alert boxes from your own debugger command files, you can use the Alert command as described below.

AddMenu Command

The AddMenu command associates commands with menu items. Its syntax is:

addmenu [*menuname* [*itemname* [*command*]]]

The *menuname* is a string expression containing the name of the menu to which to add the item. If a menu of *menuname* doesn't exist, a new menu by that name will be created.The *itemname* is a string expression that includes a command key equivalent and the name of the menu item. If a menu item with the same name exists, it is replaced; otherwise a new menu item is created. A *command* is a string expression, containing a single debugger command that will be submitted to the command interpreter when this item is selected.

If any of the optional parameters are not supplied, the current values of menus from the specified level down are displayed.

DeleteMenu Command

The DeleteMenu command removes menus or menu items. Its syntax is:

deletemenu [*menuname* [*itemname*]]

The *menuname* is a string expression containing the name of the menu from which to delete the item. The *itemname* is a string expression that contains the name of the menu item.

If only *menuname* is specified, and it contains no menu items, the menu is deleted. If a menu item with the same name specified in *itemname* exists, it is deleted.

Caution: If both *menuname* and *itemname* are omitted, all user-defined items are deleted.

Alert Command

The Alert command displays an alert box containing the specified message. The alert is displayed until the OK button is clicked. If **beep** is specified, a sound is generated when the alert box appears.

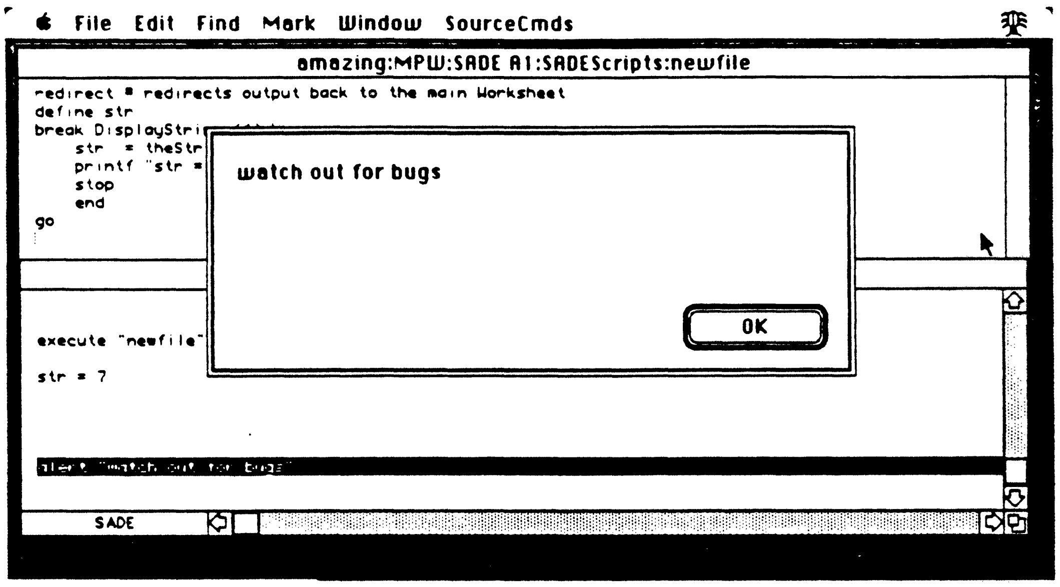
The command syntax is:

> **alert [beep]** *message*

The *message* is a string expression that will be displayed in the alert box, as in the example below:

```
alert "watch out for bugs"
```

Figure 7-2 shows the how the alert box is positioned on the screen.



```
redirect # redirects output back to the main Worksheet
define str
break DisplayStri
    str  = theStr
    printf "str =
    stop
    end
go
```

```
execute "newfile"

str = 7

alert "watch out for bugs"
```

Figure 7-2.
Position of a SADE alert box

Chapter 8
Source-Level Debugging

This chapter describes source-level debugging in SADE. It will include examples of how the user can set up his own source-level debugging world. There will be a number of screens showing the appearance of source windows and menus, including the sourceCmds menu.

- TO BE SUPPLIED

The SourceCmds menu

Right now this menu is also described in Appendix A with the other SADE menus.

Appendix A
SADE Menus

This chapter describes the SADE menus. They are similar in most respects to the MPW Apple, File, Edit, Find, Mark, and Window menus. **list any differences here***

The following sections are the same as in the MPW manual - please check to see that this information is also true of SADE. I'm sure this appendix will need extensive renovation!

Apple Menu

About SADE Displays version and copyright information.

File Menu

Each of the items in the SADE File menu is described below.

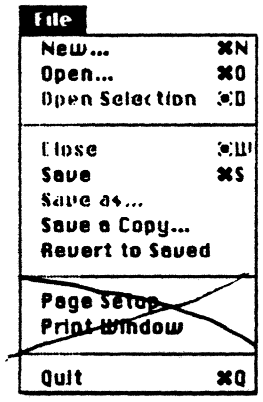


Figure A-1
File menu

NEED TO FIX THIS SCREEN DUMP

New...	Displays the New dialog box, shown in Figure A-2. The SADE New dialog box allows you to enter a name and select a directory location for the document. The Command-key equivalent is Command-N.

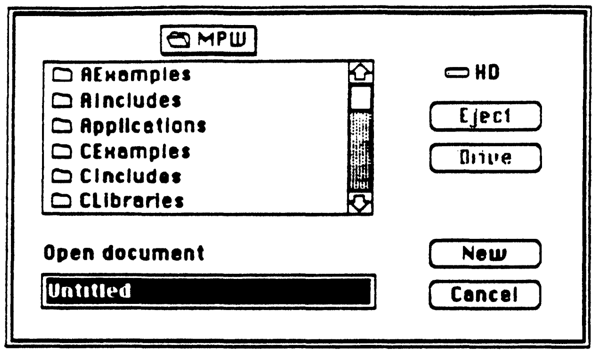


Figure A-2
New dialog box

Open...	Displays an Open dialog box (similar to that in Figure A-2) that allows you to open any TEXT file on the disk. When you open a file for the first time, the selection point is at the top of the file. When you open the file again, it reappears in the same state in which it was saved; that is, the previous selection or insertion point is preserved unless the file has been modified outside the editor. The Command-key equivalent is Command-O. *Note:* If you try to open a document that's already open in another window, that window will be brought to the front.
Open Selection	Not used in SADE.
Close	Closes the active (frontmost) window. The Command-key equivalent is Command-W.
Save	Not used in SADE.
Save as...	Displays a Save As dialog box, allowing you to change the name and directory location of the active window. Saves the current contents of the window as the "Save As" file, and allows you to continue editing the new file. The old file is closed without saving, under its original name.
Save a Copy...	Saves the current state of the active window to a new file on the disk. You can then continue editing the *old* file.
Revert to Saved	Not used in SADE.
Quit	Returns to the Finder, first allowing you to save the current state of all open files. The Command-key equivalent is Command-Q.

Edit Menu

See Appendix C for more information on using the commands on this menu.

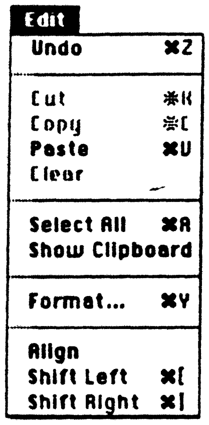
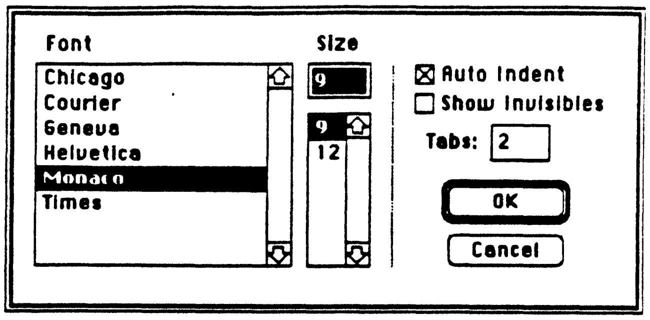


Figure A-3
Edit menu

Undo	Undoes the most recent changes to *text* in the active window (but *not* changes to resources such as font or tab settings). You can select Undo again to redo changes. The Command-key equivalent is Command-Z.
Cut	Copies the current selection in the active window to the Clipboard, and then deletes it from its original location. The Command-key equivalent is Command-X.
Copy	Copies the current selection in the active window to the Clipboard. The Command-key equivalent is Command-C.
Paste	Replaces the contents of the current selection in the active window with the contents of the Clipboard. The Command-key equivalent is Command-V.
Clear	Deletes the current selection in the active window.
Select All	Selects the entire contents of the active window. The Command-key equivalent is Command-A.
Show Clipboard	Opens a window displaying the contents of the Clipboard, if any.
Format...	Displays the Format dialog box offering a selection of fonts and sizes. The Command-key equivalent is Command-Y. This dialog box is shown in Figure A-4.



☒ Auto Indent
☐ Show Invisibles
Tabs: 2

Figure A-4
Dialog box of the Format menu item

Tabs	Sets the number of spaces that a tab character will signify for the active window. (The default tab setting is ***???***)
Auto Indent	Toggles Auto Indent on and off. When Auto Indent is on, pressing Return lines up text with the previous line. (A check mark indicates that Auto Indent is on.)
Show Invisibles	Displays the invisible characters as follows:

Tab	Δ
Space	◊
Return	¬
All other control characters	¿

The rest of the dialog box consists of a selection of the fonts installed in your System file. Available font sizes are displayed in the dialog window.

- *Note:* Selecting a font and font size affects the *entire* active window, not just the current selection in that window.

Align	Aligns the currently selected text with the top line of the selection.
Shift Left, Shift Right	These commands move selected text left or right by one tab stop. You can thus move a block of text while maintaining indentation. Shift Left adds a tab at the beginning of each line. The Command-key equivalent is Command-[. Shift Right removes a tab, or the equivalent number of spaces, from the beginning of each line. The Command-key equivalent is Command-]. If you hold down the Shift key while using these menu items, the selection will be shifted by one space, rather than by one tab.

Find Menu

Each of the items in the Find menu is described below.

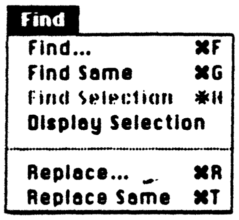
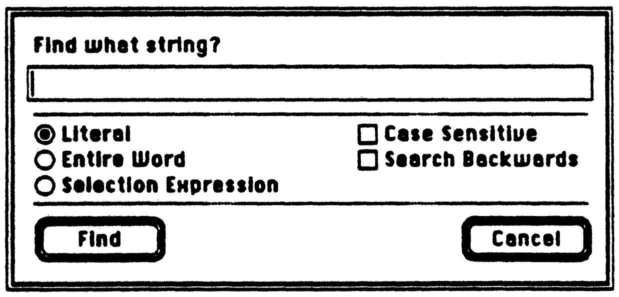


Figure A-5
Find menu

Find...	Displays a Find dialog box and finds the string you specify. By default, the Editor searches forward from the current selection in the active window (and does not wrap around). The Command-key equivalent is Command-F. This dialog box is very similar to the Find-and-Replace dialog box described below; the explanation of the radio controls and check boxes applies to both dialog boxes.
Find Same	Repeats the last Find operation, on the active window. The Command-key equivalent is Command-G.
Find Selection	Finds the next occurrence of the current selection in the active window. The Command-key equivalent is Command-H.
Display Selection	Scrolls the current selection in the active window into view.
Replace...	Displays the Find-and-Replace dialog box shown in Figure A-6 and explained below. The Command-key equivalent is Command-R.
Replace Same	Repeats the last Replace operation. The Command-key equivalent is Command-T.


Find what string?

[✓ Literal]
Entire Word
Selection Expression
☐ Case Sensitive
☐ Search Backwards

Figure A-6
Dialog box of the Replace... menu item

The operation of this dialog box is very similar to that of the Find dialog box, except that selected strings can be located and replaced with a different string throughout a file. Both dialog boxes have three radio buttons offering you one of three mutually exclusive options:

Literal	Finds the exact string (without regard for case) that you specify, wherever it may appear, even if part of other words or expressions.
Entire Word	Finds the specified string only when it occurs as a single word. To the Editor, a word is composed of the characters a–z, A–Z, 0–9, and the underscore character (_). (You can change these default values ***can you?***—see "Predefined Variables" in Chapter 3.)
Selection Expression	Enables full selection and regular expression syntax, used with the command language and described in Chapter 3. These expressions allow powerful selection and pattern matching capabilities that use a special set of metacharacters introduced below.

Any combination of the three check boxes may be selected:

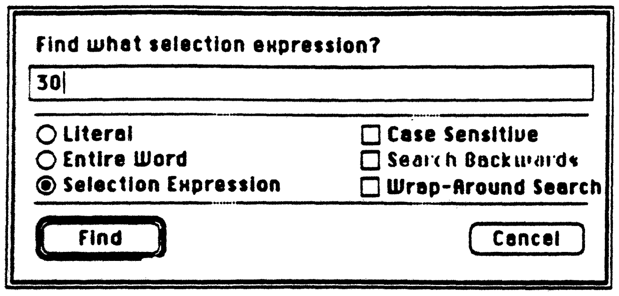
Case Sensitive	Searching is normally case insensitive; selecting this menu item specifies case-sensitive searching. (It does this ***how???***.)
Search Backward	Search backward, from the current selection to the beginning of the file. (Normally, searching is forward, and stops at the end of the file.)
Wrap-Around Search	Searches forward to the end of file, then wraps around and searches from the beginning of the file to the location of the cursor when the search was initiated. (Direction of search is reversed if Search Backward is also checked.)

- *Note:* For Find and Find-and-Replace operations, a beep indicates that the selection was not found.

Selection expressions

When the Find-and-Replace dialog's "Selection Expression" switch is selected, you can use a special set of expression operators to specify selections and text patterns. This section introduces a commonly used subset of these selection operators. Many more capabilities are available, and a full discussion can be found in Appendix C.

Selection by line number: A number given by itself specifies a line number. In the figure below, for example, the command selects line number 30 in the active window.


Find what selection expression?

30

( ) Literal
( ) Entire Word
(●) Selection Expression

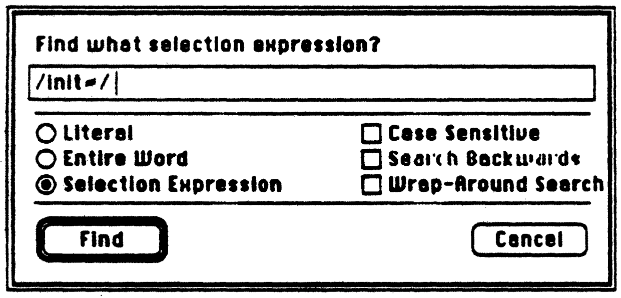
☐ Case Sensitive
☐ Search Backwards
☐ Wrap-Around Search

Figure A-7
Selection by line number

Wildcard operators: The same wildcard operators used in filename generation can also be used to specify text patterns for Find commands:

?	Any single character (other than Return).
≈	Any string of 0 or more characters, not containing a Return. (To get the ≈ character, press Option-X.)
[*characterList*]	Any character in the list. *Note:* The brackets must be typed; they don't indicate an optional syntax element.
[¬*characterList*]	Any character *not* in the list. (To get the ¬ character, press Option-L.)

These pattern matching operators are part of a larger set called **regular expression operators.** A regular expression consists of literal characters and/or regular expression operators, and must be enclosed in slashes (/.../). The figure below shows an example.


Find what selection expression?

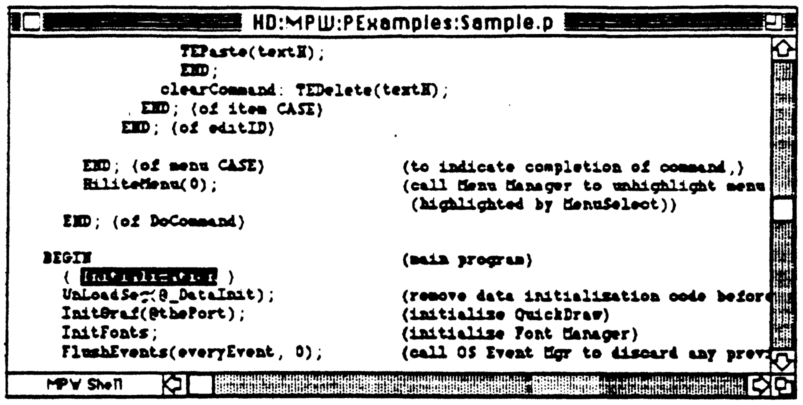
/init≈/

( ) Literal
( ) Entire Word
(●) Selection Expression

☐ Case Sensitive
☐ Search Backwards
☐ Wrap-Around Search

Figure A-8
Example of a regular expression

This command finds and selects any string that begins with "init" and is followed by any characters other than a return. Figure A-9 shows the result of this command.



```
                TEPaste(textH);
                END;
              clearCommand: TEDelete(textH);
            END; (of item CASE)
          END; (of editID)

      END; (of menu CASE)                     (to indicate completion of command,)
      HiliteMenu(0);                          (call Menu Manager to unhighlight menu
                                               (highlighted by MenuSelect))
    END; (of DoCommand)

  BEGIN                                       (main program)
    ( Initialization )
    UnLoadSeg(@_DataInit);                    (remove data initialization code before
    InitGraf(@thePort);                       (initialize QuickDraw)
    InitFonts;                                (initialize Font Manager)
    FlushEvents(everyEvent, 0);               (call OS Event Mgr to discard any prev
```

Figure A-9
Text selected with the Find command

need to change to a different display for SADE

Mark Menu

A marker is a text selection that has been given a name. Markers are useful for navigating within a window, and they can simplify many selection expressions. The upper half of the Mark menu contains the commands Mark and Unmark and the lower half lists all existing markers. To jump to the location of a marker you simply choose the name of the marker you want from the Mark menu, shown in Figure A-10 (only the marker "Here" has been created in this example).

For a detailed discussion of the syntax, characteristics, and programmatic use of markers, see Appendix C and Part II.

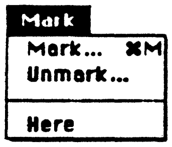


Figure A-10
Mark menu

Mark... To create a new marker interactively, first select the text you want to mark, then choose "Mark" from the Mark menu. A dialog box like that in Figure 3-11 appears, asking for the name you want the marker to have. The editable text field in the dialog box is initialized to the first word (that is, whatever you would select by a double-click) in the selection. If you click Cancel in the dialog box, the selection is unchanged and no new marker is created. If you click OK a new marker is created with the specified name and the new marker's name is added to the list of marker names displayed by the Mark menu.

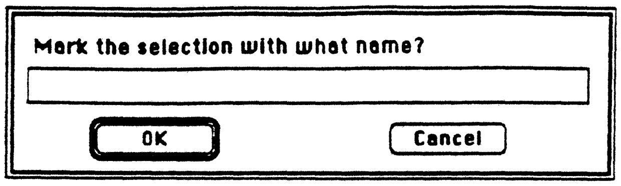


Figure A-11
Mark dialog box

If you try to create a new marker using the name of an already existing marker, a dialog box will appear, giving you the chance either to delete the old marker and add the new (OK), or to forget about adding the new marker (Cancel).

Unmark... If you choose the Unmark menu item from the Mark menu, you'll see a dialog box, like that in Figure A-12, that contains a list of currently defined markers and the two buttons Delete and Cancel. If a marker is currently selected, its name is highlighted in the marker list. You can select any number of marker names from the list. If you click Delete, every marker selected in the list is deleted. If you click Cancel, the selection remains unchanged and no markers are deleted.

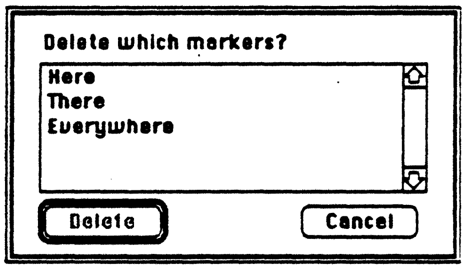


Figure A-12
Unmark dialog box

Window Menu

The upper half of the Window menu contains the two commands Tile Windows and Stack Windows; the lower half lists all open windows, as shown in Figure A-13. Selecting a window from the menu brings that window to the front, that is, superimposes it over anything else on your display. A check indicates that the window is currently the "active" window, that is, the frontmost. A bullet (•) indicates that the window is the "target" window, that is, the second to the front. Underlining indicates that a window contains changes that have not yet been saved.

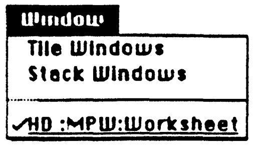


Figure A-13
Window menu

Tile Windows	Use this command to arrange windows in a tile pattern on the screen so that each window's contents are visible. Then choose a window and click its zoom box to enlarge it to full screen size.
Stack Windows	Use this command to arrange windows in a diagonally staggered pattern on your screen. This is the "open file folder" way to see several windows at once.
Worksheet	The Worksheet window always appears first in the Window menu. The menu item lists the full pathname of the worksheet.

SourceCmds Menu

The SourceCmds menu contains a group of commands implemented in the SADEStartup file. This menu is easy to customize by adding menus and menu items, following the guidelines in Chapter 8. For these menu commands to work, SADE must be able to locate the source files for your application. If the source files are not in the same directory as the application, use the Sourcepath command to specify their location.

The default contents of the SourceCmds menu include:

Break	This menu item executes the SADE proc setSourceBreak, which exists in the SADEStartup file. This proc uses the SourceToAddr function to locate the address associated with the selected location in the source file. The debugger then executes a breakpoint when the source location is reached. The Command-key equivalent is Command-B.
Unbreak	This menu item removes breakpoints set within the source file by executing the unSetSourceBreak proc. This proc exists in the SADEStartup file.The Command-key equivalent is Command-U.
Step	This command executes a single line from the source file, then halts execution. Trap calls and routines called by JSRs and BSRs will be treated as single instructions The Command-key equivalent is Command-L.

Step Into	This command executes a single line from the source file, then halts execution. Each call to a ROM routine or subroutine will cause the debugger to be reentered at the first line of the called routine. The Command-key equivalent is Command-¬.
Go	This command causes the application program to resume execution. The Command-key equivalent is Command-P.
Where PC?	This menu item highlights the line in the source file that represents the current location of the program counter. If the source file is not available, it displays an alert box with the message "Cannot find source for PC". The Command-key equivalent is Command-I.\

(***See Release Notes for current release - more options have been added, including display of variable values and switching between source level and assembly level debugging.***)

Figure A-14
SourceCmds menu

Appendix B
Sample Program

here provide listing of sample program Eventlog

Appendix C
Editing in SADE

This chapter covers what you'll need use SADE editing functions. Editing within the SADE worksheet is similar, although not identical, to the editing functions provided by the MPW worksheet. The major difference is that in SADE the editing functions are not scriptable; they are confined to the functions available from the SADE Edit menu. This chapter also includes a list of some editing shortcuts that you might like to try.

Basic editing functions are available as menu commands. You can open a file with the Open command, or by selecting its name on the screen and choosing the Open Selection command (Command-D) from the File menu. You can select and edit text with the usual Macintosh editing techniques, using menu commands to cut, copy, and paste selected text. The menu commands are described in Appendix A.

Entering Commands

By default, command output and any error messages appear in the window immediately below the executed command line. Commands are not case sensitive. You can have multiple open files, and you can enter commands in any window.

The simplest commands consist of the command name only. For example, type the command

```
version
```

and press the Enter key (without pressing Return first—that is, the insertion point must be on the same line as the command when you press Enter). This command outputs SADE version plus a timestamp:

```
Debugger (Ver 0.419) - 14:39:48 13-Apr-88
```

Commands typed into a window are referred to as **standard input.** When the results of the command(s) are then displayed in the same window (the normal, default setting) they are called **standard output.** Any window that is used to enter standard input and display standard output is referred to as the **console.**

SADE Shortcuts

Table C-1 lists some SADE shortcuts that make editing and entering commands quicker and easier. These shortcuts work in any SADE window. You can also see this list by using the SADE Help command as follows:

```
help shortcuts
```

Table C-1. SADE Shortcuts

Command	Action
Double click	select word
Triple click	select line
Double clicking before any quote (', ", or `)	select until the matching quote
Double clicking before or after {}[]()	select until the matching character
UpArrow	move selection point one line above current selection
DownArrow	move selection point one line below current selection
RightArrow	move selection point one character to the right
LeftArrow	move selection point one character to the left
CMD-Shift-UpArrow	move selection point to top of file
CMD-Shift-DownArrow	move selection point to bottom of file
CMD-DownArrow	move selection point down one screen size
CMD-RightArrow	move selection point to right edge of current line
CMD-UpArrow	move selection point up one screen size
CMD-LeftArrow	move selection point to left edge of current line
CMD-Backspace	delete from current selection to end of file
In Dialogs without an Edittext item	
Y	Yes
N	No
CMD-.	Cancel

Appendix D
Symbol File Format

here provide symbol file format

Appendix E
Object File Format

here provide explanation of compiler output

Part II - SADE Command Pages

abort	terminate break action and pending commands
addmenu	associates commands with menu items
alert	display an alert box
begin..end	group commands together
break	set breakpoint in program code
case	control case sensitivity of symbol name lookup
close	close a window
cycle	continue execution within construct
define	declare a debugger variable
deletemenu	deletes menu items
directory	write or set the default directory
disasm	disassemble and display code
dump	display unstructured memory
execute	execute debugger commands in a file
find	search for a target
for..end	looping with a control variable
func..end	define a function in the debugger language
go	resume execution
heap [display]	display heap information
heap check	verify the consistency of the heap
heap totals	display summary information for the heap
help	get help on SADE topics or commands
if..end	conditional execution of commands
kill	kills an application or tool
launch	launches an application
leave	exit from a Loop, For, While, or Repeat construct
list	lists symbols, processes, address or trap breakpoints, or tracepoints
loop	repeat commands until Leave

macro...end	associate characters with an identifier
onEntry	set commands for debugger entry
open	open file in window
printf	sends formatted output to file or window
proc...end	define a procedure in the debugger language
quit	gets out of debugger
redirect	redirect standard output
repeat...until	conditional looping with end test
resource [display]	display the resource map
resource check	check the resource map
return	exit from a proc or func
save	save specified windows
shutdown	shuts down system (with restart option)
sourcepath	identify search path for source files
stack	display stack frame
step	single step execution
stop	terminate break action
symbols	control symbolic display
target	select program target and identify symbol file
trace	sets tracepoint
unbreak	removes breakpoints.
undefine	remove definition of SADE proc, func, macros, or local variable
untrace	clears tracepoints
version	display current SADE version
while..end	repeat commands zero or more times under condition

Abort — terminate break action and pending commands

Syntax **abort**

Description The **abort** command terminates the current break action, causes the debugger to be fully entered, and cancels any debugger commands pending. This means that when the previous execution was in a structured statement, the pending commands are canceled. See also the Stop command, which terminates the break action without cancelling other pending commands.

Example

```
abort
```

See also stop, break

AddMenu — create a menu or menu items

Syntax

addmenu [*menuname* [*itemname* [*command*]]]

where

menuname is a string expression containing the name of the menu that will include the item.

itemname is a string expression containing the name of the menu item. The itemname may include a Command-key equivalent by listing the command key after a backslash ("\") at the end of the string.

command is a string expression containing a debugger command.

Description

The **addmenu** command allows you to create a menu or menu items that, when selected, will execute SADE commands. The **addmenu** parameters associate SADE commands with menu items. If a menu of *menuname* does not exist, a new menu is created. If a menu item with the same name exists, it is replaced; otherwise a new menu item is created.

The debugger command contained in the *command* parameter is submitted to the command interpreter for execution when the corresponding menu item is selected.

If any of the optional parameters are not supplied, the current values of menus from the specified level down are displayed.

Example

```
addmenu 'Debug' 'Disasm/1' 'disasm'

addmenu 'Debug' 'Code Resources' 'heap restype "CODE"'
```

See also

DeleteMenu

Alert— display an alert box

Syntax

alert [**beep**] *message*

where

message is a string expression that will be displayed in the alert box.

Description

The **alert** command displays an alert box containing the specified message. The alert is displayed until the OK button is clicked. If **beep** is specified, a sound is generatd when the alert box appears.

Example

```
alert "watch out for bugs"
```

– displays a special message

See also

to be supplied

Begin...End — group commands

Syntax

begin
  commands
end

where

  commands are one or more SADE commands

Description

The **begin** and **end** commands allow a sequence of commands to be bracketed or grouped together. One use of this construct is the specification of breakpoint actions which consist of more than one command or procedure invocation, as shown in the example below.

Output redirection applied to a **begin...end** construct will redirect the output of all the grouped commands.

Example

```
break DisplayString.(4) begin
  str := theStr^        # save value of parameter in debug
                        # variable str
  if str = '***' then
  stop
  end
end
```

See also

break , stop

Break — set a breakpoint in program code

Syntax

break *addr*, ... [*break action*]

or

break *trap range* [**from** *addr range*] [*break action*]

or

break all traps [**from** *addr range*] [*break action*]

where

addr is an address expression

break action is a single debugger command, a debugger proc call, or group of commands delimited by **begin** ... **end**.

addr range is a range expression

trap range is a address range beginning and ending with trap names of the form $†$Axxx..†$Axxx

Description

The **break** command sets one or more breakpoints within a target program's code. There are two distinct types of breakpoint: address breakpoints, and trap breakpoints. Both kinds of beakpoint may be optionally be followed by a break action, as described below.

After the desired breakpoints are set, you can then resume program execution. The program will run until it reaches the first available breakpoint, and then the debugger will stop program execution. At this point control passes back to SADE.

If the last program execution command was in a structured statement, and no break action was specified for the breakpoint, the commands immediately following the program execution command are also executed.

Address breakpoints: These can be set anywhere within program code by specifying a RAM address. Address breakpoints may also be set using symbolic references; in this case the code may not yet be in memory at the time the breakpoint is set.

Trap breakpoints: For trap ranges, either the trap name or the trap number can be used in a range expression. Trap numbers are prefixed with a "†"; for example, the range from the system trap _OpenResFile to _GetResource could be specified as

†$A997..†$A9A0

The same trap may have a breakpoint set on it by its name alone, as a member of a trap range, or as a trap within a specified address range. Each of these trap breakpoints will have a separate breakpoint record created for it; you can see what breakpoints are set using the **list break** command.

The same trap can be specified in multiple break commands. This can happen in one of two ways: either overlapping trap ranges are specified, or different address ranges are specified.

Consider the following examples where multiple break commands are set with overlapping ranges:

```
break all traps action1                # break on any trap called from any address
break †$A996..†$A9A0 action2           # break on _OpenResFile (and other traps)
                                       # called from any address
break _OpenResFile from applZone..applZone^ action3
                                       # break on _OpenResFile only if called from
                                       # application heap
break _OpenResFile from RomBase..RomBase+256K action4
                                       # break on _OpenResFile only if  called from
                                       # ROM
```

With the above trap breakpoints set, the program's calls to _OpenResFile would be handled as follows:

address	call	action
$10000	_OpenResFile	action2 is interpreted. SADE first looked for breaks on _OpenResFile only, but the address fell outside of the specified ranges. It then found the first (most recently specified) trap range that included _OpenResFile.
myproc+$100	_OpenResFile	action3 is interpreted. This call occurred within the application heap

SADE uses a search rule when finding an instance of a trap call that satisfies the conditions for a specified breakpoint. For any breakpoint that specifies a trap name, SADE first attempts to find a call within the program that exactly matches that name. Once it finds the trap, SADE checks to see if it falls into the address range if one was specified.

The next step is to check any trap ranges that might contain the trap. SADE accepts the most recently defined trap range that contains the trap, even if another trap range defined earlier also contains the trap. Once a trap range satisfying the description is found, SADE then checks to see if the range falls within the address range if one was specified.

The **break all traps** option sets a trap breakpoint on all traps in a continguous range. If *addr range* is specified, the debugger will break on traps called from the specified memory range.

Break actions: A break action can be one or more SADE commands that are meant to be interpreted after the breakpoint is reached. Normally, once SADE has performed the commands within the break action, program execution will continue. However, using a **stop** or **abort** command as part of the break action may halt subsequent program execution.

If a **stop** command is contained in a break action, and the most recent program execution command was in a structured statement, the commands immediately following the program execution command are also executed.

If an **abort** command is encountered in a break action, all pending program execution commands are aborted. The debugger is entered immediately.

Note that the commands specified in the breakpoint action are saved and not interpreted until the time when the breakpoint is reached. Consequently, any SADE variable references used in a break action should use global variables only. Local variables defined in a break action won't exist at the point when the breakpoint is reached and the break action is invoked.

Example

```
break myproc.(0) #breaks on initial statement of myproc

break _GetResource      #break on _GetResource trap

break †$A997      #another way to break on _GetResource trap

break _OpenResFile.._GetResource    #use a trap range

break †$A997..†$A9A0                #use a trap range

break all traps from DisplayText.(1)..DisplayText.(5) # breaks
# on all traps called from # DisplayText.(1)..DisplayText.(5)

break all traps from applZone..applZone^ # breaks on all traps

# called from application heap
```

See also onentry, stop, abort

Case — control case sensitivity of symbol name lookup

Syntax

case { **on** | **off** }

Description

The **case** command lets you control case sensitivity when SADE is searching for symbol names. The **case** command used with the **on** | **off** parameters works like a toggle switch to turn case sensitivity on or off.. By default, case sensitivity is turned on. Using the case command with no parameters will display the current case sensitivity.

Example

```
case on
```

- make the SADE symbol lookup case sensitive

See also

to be supplied

Close — close a window

Syntax

close [all | *windowName*]

where

windowName is a string expression specifying the file pathname of a SADE window.

Description

The **close** command closes the window for the specified file or all files. If the contents of the window are unsaved, a confirmation dialog will ask if they should be saved.

Example

```
close "myFile"
```

– close the file named "myFile"

See also

open

Cycle — continue execution within construct

Syntax

cycle [**if** *boolean expr*]

where

boolean expr is an expression

Description

The **cycle** command will cause execution to continue from the top of the currently enclosing **while**, **repeat**, **loop**, or **for** construct. If the optional **if** clause is present, the cycle action will happen only if the *boolean expr* is true, otherwise execution will continue following the **cycle** command.

Example

```
for x := 10 downto 5 do
     <commands>
     cycle if x < 2
end
```

See also

leave

Define — declare a debugger variable

Syntax

define [**global**] *var declaration* [,...]

where

var declaration has the form

name [[*dimension*]] [**:=** *init value*]

where

name must follow the rules for valid debugger identifiers, and must be unique in the current debugger scope unless declared global. The *name* may optionally be followed by an array specifier (a dimension *expr* enclosed in brackets).

dimension is an *expr*

init value is either an *expr* for the initial value of simple types, or a list of the following form for structured types:

([*expr* **of**] *init value* , ..)

where the optional **of** clause allows for replication of a value or set of values.

Description

The **define** command is used to define one or more debugger variables. Each debugger variable must be defined before it is used. A debugger variable declaration identifies the name, scope, and (optionally) the initial value of the variable. One or more variables may be declared in a single define command by having one or more *var declaration's* separated by commas.

The scope of a variable may be either global or local to the enclosing debugger proc or func. If a variable is declared at the outer level (not inside of a proc or func) then its scope is automatically global. Global variables are known both at the outer level, and inside each proc or func. If a variable is declared inside a debugger proc or func, then its scope is local unless the define command includes the **global** keyword. If a global and a local variable exist with the same name, then the local symbol overrides the global.

Redefining global variables replaces the previous definition with one exception: If the definition is within a proc or func, and the new definition matches the existing definition, the existing definition is retained. For example, a global variable definition within a proc or func creates the variable the first time the proc is invoked; subsequent invocations can make use of the value left in the variable by the preceding invocation. To remove a variable definition, use the **undefine** command.

Debugger variables are dynamically typed, that is, their type is determined on assignment (and may be changed by new assignments). The only type information supplied at definition time is whether the variable is an array or a scalar. Debugger array variables may contain a heterogeneous set of values; that is, the elements may contain values of different types.

A initial value for simple types may optionally be specified by an *expr* following an assignment operator (:=). If the item being declared is an array, a list of initial values may be specified as the values of the array elements.

Example

```
define x := 5
```

– define a debugger variable x with value equal to 5

See also undefine

DeleteMenu — delete user-defined menus or menu items

Syntax

deletemenu *menuname* [*itemname*]

where

menuname is a string expression that is the name of the menu from which to delete the item.

itemname is a string expression that is the name of a menu item.

Description

The **deleteMenu** command deletes menus and (or) menu items. If only *menuname* is specified, and it contains no menu items, the menu is deleted. If a user-defined menu item with the name specified by *itemname* exists, it is deleted. (The standard SADE menu items can't be deleted.)

Caution: If both *menuname* and *itemname* are omitted, all user-defined items are deleted.

Example

```
deletemenu "special" "launchapp"
```

– deletes the item launchapp from the special menu

See also

addMenu

Directory — set or write the default directory

Syntax

directory [*directoryname*]

where

directoryname is a string expression

Description

The **directory** command sets the default directory to the specified directory. If no directory is specified, the current default directory is written to standard output.

Example

```
directory "myOtherDir"
```

– sets default directory to myOtherDir

See also

sourcepath

Disasm — disassemble and display code

Syntax

disasm [*addr* [*count*]]

or

disasm [*addr range*]

where

addr is an address expression

count is an integer expression

addr range is a range expression

Description

The **disasm** command disassembles instructions starting at the location specified by *addr* or *addr range*. The default behavior when no address is specified is to begin disassembling at the end of the last disassembly. If the value of the program counter has changed since the last disassembly, the program counter (PC) is used as the starting address. If no range or count is specified, the number of instructions (not lines) disassembled defaults to 20.

Each line of the disassembly output is divided into four fields or areas. Their display (both order and presence) is controlled by the DisAsmFormat built-in variable as follows:

- The offset field—contains an module offset if there is one otherwise it is blank. This field is controlled by the flags 'o' and 'O'.

o	==>	display offset field in decimal.
O	==>	display offset field in hexadecimal

- The address field—contains the address of the instruction being disassembled. This field is controlled by the flags 'a' or 'A' (both have the same meaning).

A	==>	display the address

- The hex code field—contains the hexadecimal encoding for the instruction at the corresponding address. This field is controlled by the flags 'x' or 'X' (both have the same meaning).

X	==>	display the hex code representation

- The assembly code field—contains the opcode, operand, and comment disassembly for the instruction at the corresponding address. This field is controlled by the flags 'c' and 'C'.

c	==>	truncate the assembly code if necessary to a uniform length
C	==>	show entire assembly code no matter how long

The DisAsmFormat variable may also contain a '$' flag in front of the 'O', 'a', or 'A' flags to generate a "$" character in front of the offset and/or address field values.

Example

```
disasm DisplayText.(8) 5
DisplayText
+0040  003191A8  2F2D FE64   MOVE.L  -$019C(A5),-(A7)
+0044  003191AC  2F2D FE78   MOVE.L  -$0188(A5),-(A7)
+0048  003191B0  4EBA FE26   JSR     FlushDWindow    ; 00318FD8
+004C  003191B4  486D FE48   PEA     -$01B8(A5)
+0050  003191B8  4EBA 00C0   JSR     DisplayString   ; 0031927A
```

– diassembles 5 instructions in standard format, starting at the eighth statement of the DisplayText routine

See also dump

Dump — display unstructured memory

Syntax

dump [{**byte** | **<u>word</u>** | **long** }] [*addr* [*count*]]

or

dump [{**byte** | **<u>word</u>** | **long** }] [*addr range*]

where

addr is an address expression

count is an integer expression

addr range is a range expression

Description

The **dump** command displays a portion of memory at the location specified by *addr* or *addr range*. The memory is displayed in hexadecimal and ACSII characters according to the specified grouping, which may be byte, word, or long. The default grouping is **word.**

Example

```
dump DisplayText.(0)..DisplayText.(8)
00319168  4E56 FFF2 48E7 0F18  286E 0008 4AAD FE60  NV..H...(n..J..`
00319178  6700 00E8 486E FFFC  A874 2F2D FE60 A873  g...Hn...t/-.`.s
00319188  2F2D FE60 4EBA F606  266D FE74 2053 3028  /-.`N...&m.t S0(
00319198  003C 48C0 222E 000C  D280 B2AD FE68 6F14  .<H."........ho.
003191A8  2F                                        /
```

– dump memory area from within the DisplayText routine, using default word grouping

See also

disasm

Execute — execute commands in a file

Syntax

execute *filename*

where

filename is a string expression.

Description

The **execute** command lets you execute any commands contained in the specified file. An execute command can't be used within any structured statement.

Example

```
execute "myDebugCommands"
```

See also

to be supplied

Find — search for a target

Syntax

find [count] *target*[, *n*] [*addr range* [*mask*]]
or

find [count] *target*[, *n*] [*addr* [*count*] [*mask*]]

where

target is a numeric or string expression

addr range is a range expression

n is an integer expression

count is an integer expression

mask is an optional numeric expression

Description

The **find** command counts or searches program code for occurrences of a target, which may be a numeric or string expression. As options, you may start at the specified address and look up to *count* bytes beyond, or limit the search to *addr range*. The default range is HeapStart to MemTop.

The *mask* parameter is an optional numeric or string expression that is logically ANDed to the contents of each memory location before the comparison is done. The *n* parameter is an integer expression that specifies the minimum number of occurrences to ignore. In other words, the first *n*–1 occurrences may be ignored in the search. The default for *n* is 1; in this case, SADE will find the first occurrence. For example, by specifying 3 for *n*, you could find the third occurrence

Example

```
find 'mystring'
```

– searches for the string expression 'mystring'

See also

case

For — looping with control variables

Syntax

for *for clause* [**do**]
    commands
end

where *for clause* may have one of the following forms:

var **:=** *expr* **to** *expr*

var **:=** *expr* **downto** *expr*

var **:=** *expr* **,** ...

where

var is the name of a previously declared debugger variable.

expr is an expression. In the first two *for clause* forms, *expr* is an integer value. For the third form, *expr* should match the list element type of the debugger variable used.

Description

The **for** ... **end** construct provides looping with an control variable. The enclosed commands are executed until the control variable has taken on each successive value in the range expressed by the for clause.

For commands may be nested.

Example

```
For x := downto 5 do
(commands)
end
```

– do commands starting with x equal to 10 until the control variable equals 5

See also

to be supplied

Func — user-defined function

Syntax

func *name* ([*arg name*, ...])
  body
end

where

name is the function name

arg name is the argument name

body is the code that makes up the function

Description

Function definitions in the debugger language are delimited by the **func** and **end** commands. Aside from these delimiters procedure and function definitions are essentially the same. Function definitions have the additional requirement that their last statement to be executed must be a **return** command with a return value specified. The type of a function is not specified in the function definition but takes on the type of the value specified in the return statement which was executed to leave the function (which means that functions are not limited to returning results of a single type).

A **func** is called using conventional functional notation, with the function name followed by the optional actual parameters in a parenthesized list in the format of the prototype established in the **func** definition. **Func** parameters are handled in the same fashion as **proc** parameters, and the predefined debugger variables **Arg** and **NArgs** may also be used. User-defined functions may be called anywhere an *expr* is allowed.

Example

```
func fact(n)
if n <= 1.0 then
     return 1.0
else
     return n * fact(n-1)
end
end
```

– define a function factorial

See also

return

Go — resume execution

Syntax

- **go** [**til** *addr*], ..

or

go [**while** *expr*]

or

go [**until** *expr*]

where

addr is an address expression

expr is an expression

Description

The **go** command allows the debugger to resume program execution at the current program counter. The debugger sets temporary breakpoints at the specified addresses, if any. When the breakpoint is encountered, the debugger is reentered and the breakpoint is removed. If the address is in ROM, the debugger will warn you that it can't set a breakpoint in ROM. Similarly, if a condition expression is specified, the debugger will use Trace mode until the condition is met or broken (whichever is appropriate).

The debugger will be entered and control will return to the user when a breakpoint is reached, when a **while** condition becomes false, or when an **until** condition becomes true.

Example

```
go til DisplayText.(2)
```

– resume execution until the program reaches the second statement in the DisplayText routine

See also

stop

Heap — display information from heap

Syntax

heap [**display**] [*addr*] [, *blocktype*]

where

addr is an address expression

blocktype is a string expression, of which only the first 4 characters are significant. It must be one of the following values:

'purgeable' will limit the display to purgeable blocks

'nonreloc[atable]' will limit the display to nonrelocatable blocks.

'reloc[atable]' will limit the display to relocatable blocks

'free' will limit the display to free blocks.

'lock[ed]' will limit the display to locked blocks.

'res[ource] will limit the display to resources

'restype *type*' will limit the display to a resource type specified

Description

The **heap** command displays information about the specified heap objects in the current heap. If desired, you can display the heap that starts at *addr*. The default is to display the heap pointed at by theZone. By default, the information displayed is:

the address of the beginning of the heap block

the address of the master pointer if it's a relocatable block

an asterisk if the object is locked or nonrelocatable,

the value of the tag byte (for relocatables)

for a resource, the reference number of the file it's in, and the resource type and ID of the resource

You can specify one of the block types to display a subset of the heap objects. The *blocktype* must be one of the following values: purgeable, non-relocatable, relocatable, free, locked, resource, or a particular resource type.

Example

```
Heap display restype 'MENU'
BlkAddr    BlkLength Typ MasterPtr  Flags RType  RId RFRef  RName
$00316590  $00000098  H  $0031452C    R   MENU  1000 $0584  "File"
$00316838  $00000050  H  $00314528    R   MENU  1001 $0584  "Edit"
$00316888  $000000F4  H  $00314524    R   MENU  1002 $0584  "Log"
```

See also

heap check

Heap check — check consistency of the heap

Syntax

heap check [*addr*]

where

addr is an address expression

Description

The **heap check** command checks the consistency of the heap for the current target program. If desired, you can check only that part of the heap that starts at *addr*. The default is to display the heap pointed at by theZone.

The **heap check** command performs range checking to make sure all pointers are even and non-NIL, and that block sizes are within the range of the heap. It then makes sure that the self-relative handle points to a master pointer referring to the same block. For non-relocatable blocks, it checks if the heap zone pointer points to the zone where the block exists. The command also verifies that the total amount of free space is equal to the amount specified in the heap zone header, that all pointers in the free master pointer list are in master pointer blocks, and does other header validation.

Example

```
heap check
```

– checks current heap

See also

heap display

Heap totals — display heap summary

Syntax

heap totals [*addr*] [, *blocktype*]

where

addr is an address expression

blocktype is a string expression, of which only the first 4 characters are significant. It must be one of the following values:

'purgeable' will limit the display to purgeable blocks

'nonreloc[atable]' will limit the display to nonrelocatable blocks.

'reloc[atable]' will limit the display to relocatable blocks

'free' will limit the display to free blocks.

'lock[ed]' will limit the display to locked blocks.

'res[ource] will limit the display to resource blocks.

'restype *type*' will limit the display to a particular resource type

Description

The **heap totals** command displays summary information for the current heap. If desired, you may display only that part of the heap that starts at *addr*. The default is to display the heap pointed at by theZone. The summary information is given for free, nonrelocatable, and relocatable objects in the heap unless *blocktype* is specified. If *blocktype* is specified, according to the rules shown above, the summary information is limited to the indicated type of object.

Example

```
heap totals
                            Total Blks  Total Size
Free                                23       49080
Nonrelocatable                       7        1348
Relocatable                         89       21232
  Locked & NonPurgeable              2        5796
  Locked & Purgeable                 2        8136
  UnLocked & Purgeable               6         680
  UnLocked & NonPurgeable           79        6620
Heap (total)                       119       71660
```

See also

heap display

Help — Get help with SADE commands

Syntax

help [*identifier*, ...]

where

identifier is a SADE identifier

Description

The **help** command writes information about specified commands to standard output. If no command is specified, information about the **help** command is written to standard output. The search rules for the help file and the format of the help file follow those described in *The MPW Reference Manual*.

Example

```
help
```

```
SADE 1.0 Help Summaries

Help summaries are available for each of the SADE commands.
To see the list of commands enter "Help Commands". In addition,
brief descriptions of Variables, Constants, Expressions, built
in functions, and Shortcuts are also included.
```

See also

to be supplied

If...End — conditional execution of commands

Syntax

if *boolean expr* [**then**]
   commands
[**elseif** *boolean exp* [**then**]
   commands...]*..
[**else**
   commands...]...
end

where
   boolean exp is an expression
   commands are SADE commands

Description

The **if** ... **end** construct allows for conditional execution of sequences of debugger commands. Each **if** command must be followed by an **end** command. **Elseif** and **else** commands are optional, but must appear between the **if** and **end** commands in the order indicated above. More than one **elseif** may appear (indicated above by the [...]*); but at most one **else** may appear.

The commands controlled by an **if** extend to the corresponding **end**, or to the first **elseif** or **else**, whichever comes first. The commands controlled by an **elseif** extend to the next **elseif**, **else** or **end**, whichever comes first. The commands controlled by an **else** extend to the corresponding **end**.

When an **if** ... **end** construct is evaluated, if the first **if** condition (*boolean expr*) is true then the statements controlled by the **if** are executed and the remainder of the contruct to the **end** is skipped. If the condition is false the statements controlled by the **if** are skipped and the next (**elseif**) condition is checked, if present. If an **elseif** condition is evaluated and is true, then the commands it controls are executed and the remainder of the **if** construct is skipped. If no conditions were evaluated as true when the **else** command is reached (if present) then the commands controlled by the **else** are executed, otherwise they are skipped.

If ... **end** constructs may be nested.

Example

```
if x > 5 then
  <commands>
elseif x < 5 then
  <commands>
else <more commands>
end
```

– perform operations using an if...end contruct

See also to be supplied

Kill — Kill an application or tool

Syntax

kill *filename*

where

filename is a string expression

Description

The **kill** command halts the execution of the tool application specified by *filename*. Only those processes that are already suspended may be killed. This command is inherently dangerous, since an application killed with this command doesn't have the chance to perform its usual exit routines. There is no guarantee that the application's data will be saved, so use this at your own risk.

Example

```
kill "myownfile"
```

See also

launch

Launch — Launch an application

Syntax

launch *filename*

where

filename is a string expression

Description

The **launch** command launches the tool or application specified by *filename*, a string expression. This command does nothing if the filetype of the specified file is not 'APPL'. You *may* need to use the directory command before launching an application, so that SADE can locate the application.

Example

```
launch "myownfile"
```

See also

kill

Leave — exit from a Loop, For, While, or Repeat command

Syntax

leave [**if** *boolean expr*]

where

boolean expr is a boolean expression

Description

The **leave** command will cause execution to continue after the end of the currently enclosing **loop**, **while**, **repeat**, or **for** construct. If the optional **if** clause is present, the leave action will happen only if the *boolean expr* is true; otherwise execution will continue following the **leave** command.

Example

```
leave if i = 5
```

See also

loop, while, repeat, for

List — list processes, symbols, and breakpoints

Syntax

list process

or

list symbols

or

list break [{**traps** | **addrs**}]

or

list trace [{**traps** | **addrs**}]

Description

The **list** command can be used to display a list of current processes or local program symbols, as well as to display a list of address and trap breakpoints (or tracepoints).

For processes, the display includes the following information: a process number, a "loaded" or "unloaded" designation, and the filename for the process. The process numbers are incremented up to a value of 16 as each new process is started on the system. When you quit from an application, however, its process number isn't reassigned; the next process started will have a new process ID number. (See example below.)

For program symbols, the display includes the local procedure, the parent procedure, the locally-defined variables, any procedures called by the local procedure, and any types defined in the local procedure. Note that in this display, the term "module" denotes a procedure within a program.

For breakpoints and tracepoints, **list** displays the location and the symbolic representation for the location when sufficient symbolic debug information is present. If the **traps** or **addrs** modifiers are present the list will be restricted to the specified class of breakpoint. For trap breakpoints, the names of traps (or ranges of traps) with breakpoints set are displayed.

Example

```
list break
  DisplayText.(2)  ($31917C)
  DisplayText.(0)  ($319168)  <break action>
```
– display breaks currently set

```
go til DisplayText.(2)
list symbol
Module DisplayText.(0)
  Parent Module
```

```
    TransDisplay
  Variables
    theText
    len
  Contained Modules
    None.
  Types
    None.
```
–display symbols in local procedure DisplayText

```
list process
   Process#  Loaded?     FileName
          6  Loaded      "SADE"
          5  Loaded      "Microsoft Word 3.01"
          2  Loaded      "Finder"
```
–processes 3 and 4 have already been killed

See also trace, break

Loop — repeat commands until Leave

Syntax

loop
commands
end

where

commands are SADE commands

Description

The **loop** ... **end** construct provides unconditional looping. The enclosed commands are executed repeatedly. To exit the loop use the **leave** command.

Loop constructs may be nested.

Example

```
loop
(add commands here)
leave if i = 5
end
```

See also

leave

Macro — define a macro

Syntax

macro *name string expr*

where

name is an identifier

string expr is a string of characters

Description

The **macro** command allows you to define a macro, which associates a string of characters with a named identifier. When SADE encounters these characters, it interprets the macro as if it was dealing with the identifier. For instance, when the macro is used in place of a command, SADE temporarily redirects input to the command that forms the macro's actual value.

References to a macro may appear anywhere a token could appear in the debugger input stream. Macros allow the user to create abbreviations, which could be used to change the debugger language to a more terse, Macsbug-like language. (See the example below.)

Macro definitions can be nested; that is, a macro definition can contain references to other macros. Macro definitions are not recursive; references in a macro definition to the macro being expanded aren't treated as macro calls. Macros may be redefined. A macro definition is not allowed in any structured statement. Macro definitions are limited to a length of 254 characters.

Example

```
macro br 'break'
macro clr  'unbreak all'
```

See also

to be supplied

OnEntry — set commands for debugger entry

Syntax

onEntry [**break action**]

where

break action is one simple or compound SADE command

Description

The **onEntry** command can be used to execute one simple or compound SADE command each time SADE is entered. It is typically used within a startup file to provide a source display each time SADE is entered. For instance, in the SADEStartup file, the "StandardEntry" proc is the argument to the **onEntry** command. This procedure provides for source display of the current program counter (PC) on entry into the debugger.

Break actions specified using **onEntry** may also be used when entering SADE from a breakpoint. If more than one command or procedure invocation is needed within the break action, the **begin...end** constuct should be used to group a sequence of commands. A compound command formed in this way is interpreted by SADE as one unit.

Example

```
onEntry printf "%.8X\n",pc
```

- when you enter the debugger, this displays the address of the next instruction to be executed

See also

break, begin

Open — open file in window

Syntax

open [**source**] [**behind**] *filename*

where

filename is a string expression.

Description

The **open** command opens the specified file. The file must be of type 'TEXT'. If **behind** is specified, the window is opened as the window behind the frontmost SADE window, otherwise, it is opened as the frontmost window. The optional **source** modifier indicates that the window is to be treated as a special purpose source window as opposed to a general purpose text window.

Example

```
open "myFile"
```

See also

close, save

Printf — print formatted output

Syntax

printf [*format* [, *arg*] ...]

or

printf ([*format* [, *arg*] ...])

where

format is a format string with values as listed below

arg are parameters used to specify values

Description

The **printf** command places formatted output on the current output file or window. You can control SADE output using a number of different parameters with the **printf** command. These include the arg parameters and the format parameters. The arg parameters specify values to be displayed or used under control of the format string specified as the first parameter. If no format and arg parameters are specified, any buffered output is displayed.

The format string contains characters to be copied "as is" to the output and conversion specifications. Each of the format string characters applies to zero or more arg parameters. If the format is exhausted while arg parameters remain, the extra arg parameters are ignored. If there are insufficient arg parameters called for by the format, then the rest of the format string is ignored.

To distinguish a conversion specification from characters to be copied "as is" in the format string, precede it with a "%" character followed by a sequence of fields that describe how to format the arg value:

% [flags] [width] [precision] op

flags An optional sequence of characters which modify the meaning of the main conversion specification:

-	Left-justify within the field width rather than right-justify if the converted value has fewer characters than the specified minimum field width.
+	Always generate a "+" or "-" sign when converting signed arg values. Note, that negative values are always preceded by a "-" regardless of whether the "+" flag is specified.
space	Generate a space for positive values and "-" for negative values. This space is independent of any padding used to left or right-justify the value. The "+" flag has precedence over the space flag.
#	Modify the main conversion operation. The modifications performed are described in conjunction with the relevant main conversion operations discussed later.

width An optional *minimum* field width, specified as a decimal integer constant (that doesn't begin with a "0") or an "*". In the latter case a corresponding arg parameter specifies the minimum field width. If the converted value has fewer characters than the width, it will be padded to the width on the left (default) or right (if the "-" flag is specified) with spaces (default). If the converted value has more characters than the width, the width is increased to accommodate it. For %t conversions, the width specifies the minimum width to reserve for RECORD type field names.

precision The optional precision is specified as a "." followed by an *optional* decimal integer or as an "*". In the latter case a corresponding arg parameter specifies the repetition count. If the decimal integer or "*" following the "." is omitted, the precision is assumed to be 0. Precision is used to control the number of digits to be output for numeric conversions or characters for string conversions. Omitting the precision has a default value which is a function of the main conversion to be performed.

op The required main conversion operation specified as one of the following single characters:

d The corresponding arg parameter is converted to a *signed* decimal value (floating point values will be truncated).

precision The precision specifies the minimum number of digits to appear. If the value can be represented with fewer digits, leading zeros are added up to the specified precision. The result of converting a 0 value with a precision of 0 is a null. The default precision is 1.

flags	-	left-justify
	+	explicit "+" or "-"
	space	space for positive value
	#	*ignored*

u The corresponding arg parameter is converted to an *unsigned* decimal value (floating point values will be truncated).

precision The precision specifies the minimum number of digits to appear. If the value can be represented with fewer digits, leading zeros are added up to the specified precision. The result of converting a 0 value with a precision of 0 is a null. The default precision is 1.

flags	-	left-justify
	+	*ignored*
	space	*ignored*
	#	*ignored*

X The corresponding arg parameter is converted to an *unsigned hexadecimal* value. The number of bytes converted is a function of the arg's type. The letters abcdef are used for x conversion and ABCDEF are used for X conversion.

precision The precision specifies the minimum number of digits to appear. If the value can be represented with fewer digits, leading zeros are added up to the specified precision. The result of converting a 0 value with a precision of 0 is a null. The default precision is 1.

flags	-	left-justify
	+	*ignored*
	space	*ignored*
	#	prefix converted value with a "$"

b The corresponding arg parameter is converted to an *unsigned binary* value. The number of bytes converted is a function of the arg's type.

precision The precision specifies the minimum number of digits to appear. If the value can be represented with fewer digits, leading zeros are added up to the specified precision. The result of converting a 0 value with a precision of 0 is a null. The default precision is 1.

flags	-	left-justify
	+	*ignored*
	space	*ignored*
	#	*ignored*

o The corresponding arg parameter is converted to an *unsigned octal* value. The number of bytes converted is a function of the arg's type.

precision The precision specifies the minimum number of digits to appear. If the value can be represented with fewer digits, leading zeros are added up to the specified precision. The result of converting a 0 value with a precision of 0 is a null. The default precision is 1.

flags	-	left-justify
	+	*ignored*
	space	*ignored*
	#	prefix converted value with a "0"

f The corresponding arg parameter is converted to a signed decimal *floating point* value. The value is converted to the form "[-]ddd.ddd", "[-]INF", or "[-]NAN(ddd)" (where ddd is the NAN code) depending on the value.

precision The precision specifies the number of digits after the decimal point. If the precision is 0, no decimal point appears (which can be overridden with the "#" flag). The default precision is 6.

flags	-	left-justify
	+	explicit "+" or "-"
	space	space for positive value
	#	force decimal point in the case where no digits follow it

E The corresponding arg parameter is converted to a signed decimal *floating point* value. The value is converted to the form "[-]d.ddde±dd" (for e conversion), "[-]d.dddE±dd" (for E conversion), "[-]INF", or "[-]NAN(ddd)" (where ddd is the NAN code) depending on the value. The exponent will always contain at least two digits.

precision The precision specifies the number of digits after the decimal point. If the precision is 0, no decimal point appears (which can be overridden with the "#" flag). The default precision is 6.

flags	-	left-justify
	+	explicit "+" or "-"
	space	space for positive value
	#	force decimal point in the case where no digits follow it

G The corresponding arg parameter is converted to a signed decimal *floating point* value. The value is converted using f *or* e conversion (or in the style f or E conversion when G is specified). The form of conversion depends on the value being converted; e or E conversion is performed only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeros are removed from the result (which can be overridden with the "#" flag). A decimal point appears only if it is followed by a digit (which can be overridden with the "#" flag)

precision The precision specifies the *total* number of significant digits. If the precision is less than 1, then 1 is assumed. The default precision is 6.

flags	-	left-justify
	+	explicit "+" or "-"
	space	space for positive value
	#	force decimal point in the case where no digits follow it and keep trailing zeros

c The corresponding arg parameter is converted to a character (the value mod 256 is used).

precision	*ignored*	
flags	-	*ignored*
	+	*ignored*
	space	*ignored*
	#	*ignored*

s Unless the "#" flag is used, the corresponding arg parameter must be a string type (or a pointer) and the value is copied to the output as is. C strings and as is (Pascal packed array of char) strings are copied until a null is encountered (for C strings) or the number of characters specified at the precision is reached. Pascal strings may be processed if the type of the arg is a Pascal string. When the "#" flag is used, the corresponding parameter is treated as an unsigned long, and printed as if it contains 4 characters.

precision The precision specifies the maximum number of characters to output. The default precision is assumed to be infinite. In that case a C and as is strings will be output up to but not including a terminating null character and entire Pascal strings will be output

flags	-	left-justify
	+	*ignored*
	space	*ignored*
	#	the corresponding parameter is treated as an unsigned long, and printed as if it contains 4 characters

P Unless the "#" flag is used, the corresponding arg parameter must be a Pascal string type (or a pointer) and the value is copied to the output as is. When the "#" flag is used, the corresponding parameter is treated as an unsigned long, and printed as if it contains 4 characters.

precision The precision specifies the maximum number of characters to output. The default precision is assumed to be infinite. In that case the entire Pascal string will be output.

flags	-	left-justify
	+	*ignored*
	space	*ignored*
	#	the corresponding parameter is treated as an unsigned long, and printed as if it contains 4 characters

Note: You must use an upper-case %P as shown to output a Pascal string type. If you use a lower-case %p argument, the value displayed will be output as a pointer type, which is a hexadecimal number optionally preceded by 0X.

t The corresponding arg parameter is converted as a function of its type as follows:

a base type		u, d, g, p, or s as appropriate to the type with the precision and flags interpreted as a function of these format codes.
non-base type		The value(s) are displayed using a pseudo-Pascal type specification format appropriate to the type of the parameter (e.g. a RECORD/struct type is displayed using a Pascal-like RECORD notation). The flags control some of the aspects of the formatted output. Note, that the corresponding arg parameter need not specify a value and instead may specify only a type. In this case, the type definition is displayed, again using the same pseudo-Pascal type specification format.
flags	-	display only the type even if corresponding arg parameter specifies a value. The type is to be displayed exhaustively, i.e., display every type down to its base type.
	+	display only the type even if corresponding arg parameter specifies a value.
	space	show record/struct field offsets
	#	show all values and offsets in hexadecimal

% A single "%" is output; no arg is used.

precision		*ignored*
flags	-	left-justify
	+	*ignored*
	space	*ignored*
	#	*ignored*

Example

```
define i := 5
printf("fact(%.2d) = %19.19g\n", i, fact(i))
fact(05) =                 120
```

See also to be supplied

Proc — define a debugger procedure

Syntax

proc *name*[*arg name*, ...]
body
end

or

proc *name* ([*arg name*, ...])
body
end

where

name is a string expression

arg name is a SADE identifier

body is one or more SADE commands

Description

Procedure definitions in the debugger language are delimited by the **proc** and **end** commands. The **proc** command identifies the procedure's name and and an optional parameter list.

If present, the parameter list identifies parameters by name only. Parameters are not assigned a type in **proc** definitions; instead, they take on the types of their actual parameter values when the proc is called.

The parameter list may optionally be enclosed in parentheses. If the parentheses are included in the **proc** definition, they must be included when the **proc** is called as well. Similarly, if the definition was not parenthesized, then invocations of the **proc** must not be parenthesized.

Procs may be redefined. A **proc** must be defined before a call to it may be processed (so that the proc name can be recognized as such). Thus, if mutually recursive **procs** are desired, one **proc** must be defined first with a dummy **proc** definition so that the second **proc** can refer to it, and then the first **proc** can be redefined, referencing the second **proc**. The minimal dummy **proc** definition is: "**proc** foo; **end**;".

A **proc** is called by beginning a debugger command with the name of the **proc** followed by the optional actual parameter list, following the prototype in the **proc**'s definition with the actual parameter values substituted for the parameter names. The actual parameter values will be matched positionally with the formal parameter names.

The number of actual parameters need not match the number of formal parameters in the **proc** definition. If too few actual parameters are specified, the formal parameters for which there were no corresponding actual parameters will be assigned a special undefined value. Extra actual parameters have no corresponding formal name but can be referenced through the predefined debugger **Arg** array variable, which allows the parameters of a proc to be accessed positionally with references of the form "arg[n]". The number of the last actual parameter specified is contained in the predefined debugger variable **NArgs**. The values of these predefined debugger variables represent the parameter state of the currently active **proc** and are not defined outside cf the **proc**.

Proc calls may be nested.

Example

```
proc factorial n, file
define i
if nargs > 1 then
  redirect file
end
for i := 1 to n do
  printf("fact(%.2d) = %19.19g\n", i, fact(i))
end
if nargs > 1 then
  redirect
end
end
```

See also func

Quit — quit SADE

Syntax **quit**

Description The **quit** command causes the debugger to be terminated. Control returns to a process as determined by MultiFinder. Quit will display a dialog asking the user if it's all right to kill any suspended applications.

Example

```
quit
```

See also shutdown

Redirect — redirect output to file and/or window

Syntax

redirect [**append**] *filename*

or

redirect [**pop**] [**all**]

where

filename is a string expression

Description

The **redirect** command redirects the output from SADE commands to the specified file. If you specify the **append** parameter, the output will be appended to the end of the file.

You may nest Redirect commands to as many as 10 different files; SADE will maintain the names of these files as a last-in first-out queue. If you use the **pop** parameter, or if you use no parameters at all with the Redirect command, the output from SADE commands is redirected to the file at the head of the queue. If **all** or **pop all** is specified, standard output is redirected to the SADE WorkSheet.

Note: any error conditions cause SADE to perform an implicit **pop all** for any redirected files. This ensures that output will return to the SADE WorkSheet.

Example

```
redirect "myOutputFile"
```

- redirect stdout to myOutputFile

See also

to be supplied

Repeat...until — conditionally repeat commands

Syntax

repeat
commands
until *boolean expr*

where

commands are SADE commands
boolean expr is an expression

Description

The **repeat** ... **until** construct provides conditional looping with a test at the end of the loop. The enclosed commands are executed until *boolean expr* is true. The enclosed commands are executed at least once.

Repeat constructs may be nested.

Example

```
repeat
(supply commands here)
until x = 5
```

See also

leave

Resource — Display the resource map

Syntax

resource [**display**] [*addr*] [**restype** '*type*']

where

addr is an address expression

type is a valid resource type

Description

The **resource** command displays the resource map at *addr*. The default, if no address is specified, is to display all resource maps for the target application. The information displayed for each map includes: its location, the refnum of the resource file, and a list of the instances of each type. For each resource type displayed within the map, the following information is displayed: the resource ID, the resource type, the value of the master pointer, whether the resource is locked or unlocked, and the resource name.

You can also restrict the resource display to a particular resource type, using the **restype** '*type*' designation. See the example below.

Example

```
resource restype 'WIND'
Resource Map at $00316EF8
   ResId  RType  MasterPtr  Locked?   Name
    1000  WIND   $00316BE4  Unlocked
    1001  WIND   $00316C08  Unlocked
    1002  WIND   $00316484  Unlocked
    1003  WIND   $003164B8  Unlocked
    1004  WIND   $003164D8  Unlocked
Resource Map at $0002B19C
   ResId  RType  MasterPtr  Locked?   Name
  -16000  WIND   NotLoaded
  -15968  WIND   NotLoaded
  -15840  WIND   NotLoaded
```

See also

resource check

Resource check — check the resource map

Syntax

resource check [*addr*]

where

addr is an address expression

Description

The **resource check** command checks the resource map at *addr* for consistency. The default, when no address is specified, is to validate all resource maps for the target application. If an inconsistency is found, the command displays a diagnostic message specifying the problem.

Example

```
resource check

    Resource map is okay.
```

See also

resource display

Return — exit from a procedure or function

Syntax

return [*expr*]

where

expr is a function value

Description

The **return** command causes the debugger to exit the debugger procedure or function currently in execution. When **return** is used to return from a function, an expression must be specified for the function value. When returning from procedures there should be no return value.

Example

```
return
```

See also

func, proc

Save — save specified window(s)

Syntax

save [**all** | *filename*]

where

filename is a string expression.

Description

The **save** command saves the specified file or, if **all** is specified, saves all files. The default file is the currently selected window. If the file was not modified since the last time it was saved, nothing within the file is affected by this command.

Example

```
save "myFile"
```

See also

open, close

Shutdown — power down or restart the machine

Syntax **shutdown** [**restart**]

Description The **shutdown** command lets you shut down your system from within SADE. When you use the shutdown command, it first causes the debugger process to be terminated, which is the same as issuing a quit command. In addition, the shutdown command calls the ShutDown Manager, which in turns closes all other applications and shuts down the system. If **restart** is specified as an option, the Macintosh will be restarted.

Example

```
shutdown
```

See also quit

Sourcepath — identify search path for source file display

Syntax

sourcepath [[**add** | **del**[**ete**]] *directoryname*, ...]

where

directoryname is a string expression

Description

The **sourcepath** command is used to affect the search path of directories used for source file display. The specified *directoryname* indicates where the AddrToSource function should look to find files for source display. You can use a list of directory names to allow the use of source files in more than one directory. If no directory is specified, the current search path of directories is written to standard output. The **add** and **delete** options allow particular directories to be added (to the end) and removed from the search path.

Example

```
sourcepath 'srcdir', :myotherdir'  # sources in more than one
                                   # directory
sourcepath add ":samples"          # add the directory Samples
                                   # to the search path
```

See also

directory

Stack — display stack frames

Syntax

stack [*count*] [**at** *addr*]

where

count is an expression

addr is an address expression

Description

The **stack** command displays a list of the stack frames for the current target application. The stack frames displayed are based on register A6 or *addr* if **at** is specified.

For each stack frame, the location of the frame pointer is indicated by the frame address. The frame owner corresponds to the portion of the program that used the frame. The procedure that called the frame is listed with an offset if needed.

If an explicit *count* is specified, then at most that many stack frames will be displayed.

Example

```
stack at DisplayText.(6)
stack
 Frame Addr  Frame Owner   Called From
  <main>      CMain
  $0032BC24   main          CMain+$0028
  $0032BB2C   SkelMain      main.(51)
  $0032BB0C   LogEvent      SkelMain.(13)+$0012
  $0032BADC   ReportUpdate  LogEvent.(50)+$0004
  $0032BACC   DisplayText   ReportUpdate.(1)+$0004
```

See also

to be supplied

Step — single step execution

Syntax

step [{**asm** | **line**}] [**into**]

Description

The **step** command is used for single step execution of the target program. This allows the user to execute one line of code at a time. If **line** (the default) is specified, the debugger will execute all of the instructions associated with the current source line. If **into** is specified, each call to a ROM routine or subroutine will cause the debugger to be reentered at the first instruction (line) of the called routine. Otherwise, trap calls and routines called by JSRs and BSRs will be treated as single instructions.

If **asm** is specified, the debugger executes the instruction at the current program counter location. The debugger is reentered after the instruction executes.

If the source window containing the current line is displayed, the next line to be executed is indicated graphically.

Example

```
step
```

See also

to be supplied

Stop — terminate break action

Syntax **stop**

Description The **stop** command terminates the current break action in an executing application program. If the previous execution was in a structured statement, the command following the execution command is completed. See also the Abort command, which terminates the break action and all other pending commands as well.

Example

```
stop
```

See also abort, quit, break

Symbols — control symbol display

Syntax **symbols** [{**on** | **off**}]

Description The **symbols** command permits control over symbol display within SADE windows. By default, debugger command output is always displayed symbolically whenever possible. Setting symbols off will disable attempts to display symbolic representations of debugger command outputs. This might be desirable for speed. If the symbol command is used with no argument, the current state is written to standard output.

Example

```
symbols on
```

- turns symbols on

See also to be supplied

Target — select program and identify symbol file

Syntax

target *progname* [**using** *symbolfilename*]

where

progname is a string expression

symbolfilename is a string expression

Description

The **target** command selects the program for which all subsequent SADE commands and symbolic references are intended. The optional **using** parameter should be specified if the name of the symbol file for the program is not *progname*.sym in the same directory as the program.

Example

```
target "myProgram" using ":anotherdir:myprogram.sym"
```

See also

to be supplied

Trace — set tracepoints

Syntax

trace *addr*,...

or

trace *trap range* [**from** *addr range*], ..

or

trace all traps [**from** *addr range*]

where

addr is an address expression

trap range is a range expression of the form †$Axxx..†$Axxx

addr range is a range of address locations

Description

The **trace** command sets tracepoints on the specified address or traps within the target application. After setting the tracepoints, you can resume program execution. When the tracepoint is encountered in the executing program, a message is displayed on the current standard output, reporting the address or trap being traced, and optionally the symbolic representation of the address. If *addr range* is specified, the message will be displayed only if the trap was called from the specified memory range. In any case, program execution is resumed after the message is displayed.

For trap ranges, either the trap name or the trap number can be used in a range expression. Trap numbers may be prefixed with a "†"; for example, the range from the system trap _OpenResFile to _GetResource could be specified as

†$A997..†$A9A0

Example

```
trace _OpenResFile.._GetResource     #use a trap range
trace †$A997..†$A9A0                 #use a trap range
```

See also

untrace

Unbreak — remove breakpoints

Syntax

unbreak *addr*, ...

or

unbreak *trap range*, ...

or

unbreak all [{**traps** | **addrs**}]

where

addr is an address expression

trap range is a range of trap locations

Description

The **unbreak** command clears the breakpoint set for the specified address or addresses, or for a range of traps. In addition, it removes any break actions associated with cleared breakpoints. The **all** form of the command clears all breaks set in the target program. The **break all** form can optionally restricted to just traps or addresses if the **traps** or **addrs** modifiers are present.

Example

```
unbreak _GetResource   #undo break on GetResource trap
```

See also

break, break all traps

Undefine — remove definitions of SADE procs, funcs, or macros

Syntax

undefine *identifer*, ...

where

identifier is a SADE identifier representing a proc, func, or macro name

Description

The **undefine** command removes the definition of the specified global SADE variable, proc, func, or macro. A list of identifers can also be used to remove more than one definition. This command allows you to get rid of a global SADE variable, proc, func, or macro name that is no longer needed. Note that this doesn't remove variables defined within a SADE proc or func.

If you want to redefine a variable, proc, func, or macro name, you don't need to use **undefine**; you can just assign a new value to the existing name.

Example

```
undefine id   #remove the macro definition for id (diassemble)
```

See also

proc, func, macro, define

Untrace — remove tracepoints

Syntax

untrace *addr*,...

or

untrace *trap range* ,

or

untrace all [{**traps** | **addrs**}]

where

addr is an address expression

trap range is a range of trap locations

Description

The **untrace** command clears the tracepoint at the specified addresses or traps. The **all** form clears all tracepoints within the target program. The **untrace all** form can optionally be restricted to just traps or addresses if the **traps** or **addrs** modifiers are present.

Example

```
untrace _GetResource   #undo trace on _GetResource trap
```

See also

trace

Version — Display SADE version information

Syntax **version**

Description The **version** command displays the current SADE version number and the current time and date.

Example

```
version

Debugger (Ver 0.419) - 11:37 16-Mar-88
```

See also to be supplied

While...End — repeat commands zero or more times under condition

Syntax

while *boolean expr* [**do**]

  commands

end

where

  boolean expr is a boolean expression

  commands are SADE commands

Description

The **while** ... **end** construct provides conditional looping. The enclosed commands are executed as long as *boolean expr* is true. If the condition is false, the enclosed commands are skipped and execution resumes following the **end**.

While constructs may be nested.

Example

```
while i = 5 do
(supply commands here)
end
```

See also

leave

Changes from MPW 3.0A1

Pascal

Bug fixes for heap management and text file I/O.

CLib

printf and scanf now conform to ANSI standard:

- %p now means write (read) the value of a pointer.
- Pascal strings are now written (read) by the %P specifier.
- %n means to write into the specified location the number of characters written (read) so far.
- Extended values are read by using the character 'L' as a modifier with %e, %f, or %g. 'n' is no longer supported for this purpose; i.e. use %Le instead of %ne, etc.
- %i is now implemented for both printf and scanf. In printf, it is equivalent to %d. In scanf, it reads an octal, hexadecimal, or decimal integer; it determines the base using the normal C conventions. (For example, 10, 010, and 0x10 correspond to base 10 values of 10, 8, and 16, respectively.)

Known bugs and limitations:

- The new ANSI header file limits.h was inadvertently omitted from this release.

ANSI functions new in this release are:

• § 4.10.4.5	system()
• § 4.10.7.1	mblen()
• § 4.10.7.2	mbtowc()
• § 4.10.7.3	wctomb()
• § 4.10.8.1	mbstowcs()
• § 4.10.8.2	wcstombs()
• § 4.11.2.2	memmove()
• § 4.11.5.7	strstr()

The following descriptions of ANSI functions are taken from the May 13, 1988, draft of the standard.

4.10.4.5 The system function

Synopsis

```
#include <stdlib.h>
int system(const char *string);
```

Description

The system function passes the string pointed to by string to the host environment to be executed by a command processor in an implementation-defined manner. A null pointer may be used for string to inquire whether a command processor exists.

Returns

If the argument is a null pointer, the system function nonzero only if a command processor is available. If the argument is not a null pointer, the system function returns an implementation-defined value.

4.10.7.1 The mblen function

Synopsis

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

Description

If s is not a null pointer, the mblen function determines the number of bytes comprising the multibyte character pointed to by s. Except that the shift state of the mbtowc function is not affected, it is equivalent to

```
        mbtowc ((wchar_t *) 0, s, n);
```

The implementation shall behave as if no library function calls the mblen function.

Returns

If s is a null pointer, the mblen function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If s is not a null pointer, the mblen function either returns 0 (if s points to the null character), or returns the number of bytes that comprise the multibyte character (if the next n or fewer bytes form a valid multibyte character), or returns -1 (if they do not form a valid multibyte character).

4.10.7.2 The mbtowc function

Synopsis

```
        #include <stdlib.h>
        int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Description

If s is not a null pointer, the mbtowc function determines the number of bytes that comprise the multibyte character pointed to by s. It then determines the code for value of type wchar_t that corresponds to that multibyte character. (The value of the code corresponding to the null character is zero.) If the multibyte character is valid and pwc is not a null pointer, the mbtowc function stores the code in the object pointed to by pwc. At most n bytes of the array pointed to by s will be examined.

The implementation shall behave as if no library function calls the mbtowc function.

Returns

If s is a null pointer, the mbtowc function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If s is not a null pointer, the mbtowc function either returns 0 (if s points to the null character), or returns the number of bytes that comprise the converted multibyte character (if the next n or fewer bytes form a valid multibyte character), or returns -1 (if they do not form a valid multibyte character).

In no case will the value returned by greater than n or the value of the MB_CUR_MAX macro.

4.10.7.3 The wctomb function

Synopsis

```
        #include <stdlib.h>
        int wctomb(char *s, wchar_t wchar);
```

Description

The wctomb function determines the number of bytes needed to represent the multibyte character corresponding to the code whose value is wchar (including any change in shift state). It stores the multibyte character representation in the array object pointed to by s (if s is not a null pointer). At most MB_CUR_MAX characters are stored. If the value of wchar is zero, the wctomb function is left in the initial shift state.

The implementation shall behave as if no library function calls the wctomb function.

Returns

If s is a null pointer, the wctomb function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If s is not a null pointer, the wctomb function returns -1 if the value of wchar does not correspond to a valid multibyte character, or returns the number of bytes that comprise the multibyte character corresponding to the value of wchar.

In no case will the value returned by greater than n or the value of the MB_CUR_MAX macro.

4.10.8.1 The mbstowcs function

Synopsis

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

Description

The mbstowcs function converts a sequence of multibyte characters that begins in the initial shift state from the array pointed to by s into a sequence of corresponding codes and stores not more than n codes into the array pointed to by pwcs. No multibyte characters that follow a null character (which is converted into a code with value zero) will be examined or converted. Each multibyte character is converted as if by a call to bye mbtowc function, except that the shift state of the mbtowc function is not affected.

No more than n elements will be modified in the array pointed to by pwcs. If copying takes place between objects that overlap, the behavior is undefined.

Returns

If an invalid multibyte character is encountered, the mbstowcs function returns (size_t) -1. Otherwise, the mbstowcs function returns the number of array elements modified, not including a terminating zero code, if any.

[footnote: The array will not be null- or zero-terminated if the value returned is n.]

4.10.8.2 The wcstombs function

Synopsis

```
#include <stdlib.h>
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

Description

The wcstombs function converts a sequence of codes that correspond to multibyte characters from the array pointed to by pwcs into a sequence of multibyte characters that begins in the initial shift state and stores these multibyte characters into the array pointed to by s, stopping if a multibyte character would exceed the limit of n total bytes or if a null character is stored. Each code is converted as if by a call to the wctomb function except that the shift state of the wctomb function is not affected.

No more than n bytes will be modified in the array pointed to by s. If copying takes place between objects that overlap, the behavior is undefined.

Returns

If a code is encountered that does not correspond to a valid multibyte character, the wcstombs function returns (size_t) -1. Otherwise, the wcstombs function returns the number of bytes modified, not including a terminating null character, if any.

[footnote: The array will not be null- or zero-terminated if the value returned is n.]

4.11.2.2 The memmove function

Synopsis

```
        #include <string.h>
        void memmove(void *s1, const void *s2, size_t n);
```

Description

The memmove function copies n characters from the object pointed to by s2 into the object pointed to by s1. Copying takes place as if the n characters from the object pointed to by s2 are first copied into a temporary array of n characters that does not overlap the objects pointed to by s1 and s2, and then the n characters from the temporary array are copied into the object pointed to by s1.

Returns

The memmove function returns the value of s1.

4.11.5.7 The strstr function

Synopsis

```
        #include <string.h>
        char *strstr(const char *s1, const void *s2);
```

Description

The strstr function locates the first occurrence in the string pointed to by s1 of the sequence of characters (excluding the terminating null character) in the string pointed to by s2.

Returns

The strstr function returns a pointer to the located string, or a null pointer if the string is not found. If s2 points to a string with zero length, the function returns s1.

The C Libraries have the following changes from the ERS.

Text and Binary Streams

Text streams are not yet supported; all files are opened as if binary mode had been requested. This feature will not be implemented pending a resolution of the \n vs. \r controversy.

New ANSI functions implemented in this release; manual pages for these functions appear below.

- remove()
- rename()
- tmpnam()
- vfprintf()
- vprintf()
- vsprintf()
- fgetpos()
- fsetpos()
- perror()
- bsearch()
- strerror()

Functions/facilities changed for ANSI compatibility

- VarArgs.h changed to StdArg.h; portable mechanism for using optional arguments changed.

```
      Pre-ANSI style:
        /*
         *  Sample function declaration taking 2 or more arguments.
         *  Key macro definitions are in VarArgs.h.
         */
        #include <VarArgs.h>

        /*
         *  By convention, the function declaration uses va_alist as the name
         *  of the first optional argument.  The header file VarArgs.h defines the
         *  macro va_dcl, which is used as the declaration of va_alist.  Note that
         *  no semicolon follows va_dcl.
         */
        int foo(arg1, arg2, va_alist)
        char *arg1;
        int arg2;
        va_dcl
        {
          /*
           *  A local variable, nextArg, is used to indicate the current position
           *  in the list of optional arguments.  The type, va_list is defined in
           *  the header file StdArg.h.
           */
          va_list nextArg;

          /*
           *  The macro va_start, defined in VarArgs.h, initializes its argument to refer to
           *  the first optional argument passed to the function.
           */
          va_start(nextArg);

          ...

          /*
           *  To refer to each optional argument in turn, both the current position and
           *  the type of the argument must be passed to the macro va_arg, which is defined
           *  (of course) in VarArgs.h.
           */
          va_arg(nextArg, int);

          ...
        /*
         *  The macro va_end does any necessary cleanup from using optional arguments.
         *  It is defined, to our great surprise, in VarArgs.h.
         */
        va_end(nextArg);
      }


    ANSI style:
      /*
       *  Sample function declaration taking 2 or more arguments.
       *  Key macro definitions are in StdArg.h.
       */
      #include <StdArg.h>

      /*
       *  ANSI function declarations use ', ...' to indicate the position
       *  of the first optional argument.
       */
      int foo(arg1, arg2, ...)
      char *arg1;
      int arg2;
      {
        /*
         *  A local variable, nextArg, is used to indicate the current position
         *  in the list of optional arguments.  The type, va_list is defined in
         *  the header file StdArg.h.
         */
        va_list nextArg;

        /*
         *  The macro va_start, defined in StdArg.h, takes two arguments: the
         *  variable used to refer to the optional arguments, and the name of
         *  the last named argument passed to the function.
         */
        va_start(nextArg, arg2);

        ...

        /*
         *  To refer to each optional argument in turn, both the current position and
         *  the type of the argument must be passed to the macro va_arg, which is defined
         *  (of course) in StdArg.h.
         */
        va_arg(nextArg, int);

        ...

        /*
         *  The macro va_end does any necessary cleanup from using optional arguments.
         *  It is defined, to our great surprise, in StdArg.h.
         */
        va_end(nextArg);
      }
```

Descriptions of the newly-implemented ANSI functions, taken from the January, 1988, draft proposed standard.

4.9.4.1 The remove function

Synopsis

```
    #include <stdio.h>
    int remove(const noalias char *filename);
```

Description

The remove function causes the file whose name is the string pointed to by filename to be no longer accessible by that name. A subsequent attempt to open that file using that name will fail, unless it is created anew. If the file is open, the behavior of the remove function is implementation-defined.

Returns

The remove function returns zero if the operation succeeds, nonzero if it fails.

4.9.4.2 The rename function

Synopsis

```
    #include <stdio.h>
    int rename(const noalias char *old, const noalias char *new);
```

Description

The rename function causes the file whose name is the string pointed to by old to be henceforth known by the name given by the string pointed to by new. The file named old is effectively removed. If a file named by the string pointed to by new exists prior to the call to the rename function, the behavior is implementation-defined.

Returns

The rename function returns zero if the operation succeeds, nonzero if it fails, in which case if the file existed previously it is still known by its original name.

[footnote omitted]

4.9.4.4 The tmpnam function

Synopsis

```
    #include <stdio.h>
    char *tmpnam(noalias char *s);
```

Description

The tmpnam function generates a string that is not the same as the name of an existing file.

The tmpnam function generates a different string each time it is called, up to TMP_MAX times. If it is called more than TMP_MAX times, the behavior is implementation-defined.

The implementation shall behave as if no library function calls the tmpnam function.

Returns

If the argument is a null pointer, the tmpnam function leaves its result in an internal static object and returns a pointer to that object. Subsequent calls to the tmpnam function may modify the same object. If the argument is not a null pointer, it is assumed to point to an array of at least L_tmpnam characters; the tmpnam function writes its result in that array and returns the argument as its value.

Environmental limits

The value of the macro TMP_MAX shall be at least 25. [In our implementation, TMP_MAX is 17576.]

[footnote omitted]

4.9.6.7 The vfprintf function

Synopsis

```
    #include <stdarg.h>
    #include <stdio.h>
    int vfprintf(FILE *stream, const noalias char *format,
                 va_list arg);
```

Description

The vfprintf function is equivalent to fprintf, with the variable argument list replaced by arg, which has been initialized by the va_start macro (and possibly subsequent va_arg calls). The vfprintf function does not invoke the va_end macro.

Returns

The vfprintf function returns the number of characters transmitted, or a negative value if an output error occurred.

[example omitted]

4.9.6.8 The vprintf function

Synopsis

```
    #include <stdarg.h>
    #include <stdio.h>
    int vprintf(const noalias char *format, va_list arg);
```

Description

The vprintf function is equivalent to printf, with the variable argument list replaced by arg, which has been initialized by the va_start macro (and possibly subsequent va_arg calls). The vprintf function does not invoke the va_end macro.

Returns

The vprintf function returns the number of characters transmitted, or a negative value if an output error occurred.

4.9.6.8 The vsprintf function

Synopsis

```
    #include <stdarg.h>
    #include <stdio.h>
    int vsprintf(noalias char *s, const noalias char *format,
                 va_list arg);
```

Description

The vsprintf function is equivalent to sprintf, with the variable argument list replaced by arg, which has been initialized by the va_start macro (and possibly subsequent va_arg calls). The vsprintf function does not invoke the va_end macro.

Returns

The vsprintf function returns the number of characters written in the array, not counting the terminating null character.

4.9.9.1 The fgetpos function

Synopsis

```
    #include <stdio.h>
    int fgetpos(FILE *stream, noalias fpos_t *pos);
```

Description

The fgetpos function stores the current value of the file position indicator for the stream pointed to by stream in the object pointed to by pos. The value stored contains unspecified information usable by the fsetpos function for repositioning the stream to its position at the time of the call to the fgetpos function.

Returns

If successful, the fgetpos function returns zero; on failure, the fgetpos function returns nonzero and stores an implementation-defined positive value in errno.

4.9.9.3 The fsetpos function

Synopsis

```
    #include <stdio.h>
    int fsetpos(FILE *stream, const noalias fpos_t *pos);
```

Description

The fsetpos function sets the file position indicator for the stream pointed to by stream according to the value of the object pointed to by pos, which shall be a value returned by an earlier call to the fgetpos function on the same stream.

A successful call to the fsetpos function clears the end-of-file indicator for the stream and undoes any effects of the ungetc function on the same stream. After an fsetpos call, the next operation on an update stream may be either input or output.

Returns

If successful, the fsetpos function returns zero; on failure, the fsetpos function returns nonzero and stores an implementation-defined positive value in errno.

4.9.10.4 The perror function

Synopsis

```
    #include <stdio.h>
    void perror(const noalias char *s);
```

Description

The perror function maps the error number in the integer expression errno to an error message. It writes a line to the standard error stream thus: first (if s is not a null pointer and the character pointed to by s is not the null character), the string pointed to by s followed by a colon and a space; then an appropriate error message string followed by a new-line character. The contents of the error message strings are the same as those returned by the strerror function with argument errno, which are implementation-defined.

Returns

The perror function returns no value.

4.10.5.1 The bsearch function

Synopsis

```
    #include <stdlib.h>
    void *bsearch(const noalias void *key,
                  const noalias void *base,
                  size_t nmemb, size_t size,
                  int (*compar) (const noalias void *,
                                 const noalias void *));
```

Description

The bsearch function searches an array of nmemb objects, the initial member of which is pointed to by base, for a member that matches the object pointed to by key. The size of each member of the array is specified by size.

The contents of the array shall be in ascending sorted order according to a comparison function pointed to by compar, which is called with two arguments that point to the key object and to an array member, in that order. The function shall return an integer less than, equal to, or greater than zero if the key object is considered, respectively, to be less than, to match, or to be greater than the array member.

Returns

The bsearch function returns a pointer to a matching member of the array, or a null pointer if no match is found. If two members compare as equal, which member is matched is unspecified.

[footnote omitted]

4.11.6.2 The strerror function

Synopsis

```
    #include <string.h>
    char * strerror(int errnum);
```

Description

The strerror function maps the error number in errnum to an error message string.

Returns

The strerror function returns a pointer to the string, the contents of which are implementation-defined. The array pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the strerror function.

Release Notes for MPW 3.0A2 - Linker and related tools

Link

A2 Features

- Added -ad size and -ac size options to align data and code modules on power-of-two boundaries. The size argument must be a power of two (checked). Note that Pascal emits global variables as entry points in a large module, so typically only the last variable in the source (the variable with the largest A5-offset) will be aligned as specified.
- The output of "-rt", other than CODE=0, is limited to one resource. It's an error to generate more than one segment if there is no jump table.
- A crude form of type equivalence now takes place. Two types are considered identical (and are coelesced) if they have the same type name and were defined in the same location in the same source file. Before this, if you had #included, say, QuickDraw.h in each of your 80 C source files, there would have been 80 copies of the QuickDraw type information in the symbolic output file.
  It's possible to confuse the equivalence requirements, but the situtations appear unlikely. For instance:
  ```
  type.h              file1.c               file2.c
  struct bar {        #define XXX short     #define XXX long
    XXX field;        #include "type.h"     #include "type.h"
  };
  ```
  The body of the structure "bar" is determined by a macro expansion, which results in two different types with the same name and source location.
- Object files containing symbolic information must have a version number of 3 in the First record. Link and Lib will emit warnings (for Beta) and errors (for Final) if symbolics are encountered in version 1 or 2 files.

Pre-A2 Features

- The Linker has been enhanced to support SADE information in the form of a symbolic information file. This file will be named "*filename*.SYM" where *filename* is the name of the linker output file.
- A new option -SYM has been added to the Linker for SADE support. If this option is specified, one or more parameters must also be supplied.
- The available parameters are ON, OFF, FULL, NOVARS, NOLINES, and NOTYPES. Parameters are processed left to right, and if thus may interact unfavourably since there is currently no consistency check done.
- ON and FULL are currently equivalent and are provided solely to conform to specifications provided by the SADE team. NOVARS will suppress the symbolic information for variables, NOLINES will suppress source statement information, and NOTYPES will suppress type description information.
- In a subsequent release, FULL in juxtaposition with NO... will generate an error.

- If -SYM is not specified, symbolic information will not be processed.
- The OMF has been changed substantially to accomodate symbolic debugging information. (See the OMF appendix in the manual.)
- Other Option Changes
  Both the -LA and the -LF options now imply -L.
  The -B, -BF and -BS options have become obsolete and should be removed from build scripts. They are flagged as warnings with the Alpha release but will generate errors on future releases.
- Large Global Data Areas
  In response to popular demand the MPW 3.0 Linker supports global data segments > 32K.
  Not so fast! In order to use large data areas, it is necessary for the object code emitted by compilers/assemblers to reference global data with 32-bit A5 offsets. (In MPW 3.0 C this is accomplished by specifying "-fx 1" on the command line.) Unfortunately, libraries are built using 16 bit offsets. It becomes the responsibility of the user to order the link input so that data which are referenced with 16 bit offsets are within 32K of A5.
  Since the linker allocates data from the bottom up the data modules encountered first are allocated furthest from A5 - with one exception: a data module with the "main module" bit set will be located next to A5.
  So - caveat user.
  In addition, if the data segment is to be larger than 32k , this release requires that the "-ss" option be used to specify the allowable size of the largest segment.

Lib

- Understands symbolic information (it's no longer stripped). The default is to keep symbolics ("-Sym On"). You can strip symbolic information completely by specifying "-Sym Off", or you can strip subsets of the information (e.g. "-Sym NoVars"). Object files without symbolic information are unaffected.
  [You can't strip types in A2 because of a bug in "-Sym NoTypes". Beta.]
- The -B, -BF, -BS options are now obsolete. They will be removed in subsequent releases, so change scripts to remove these options.

DumpObj

- If -m is given the name of an entry point, DumpObj backs up to the beginning of the Module in which the entry point is defined before dumping or disassembling.
- In the disassembled code, the display now resolves references to show actual reference names when possible, and also appends the ID number parenthetically following ID names. Furthermore, the anachronistic block and byte offsets produced with the -L option has been changed to reflect MPW file offsets.
- Of course, DumpObj now supports the display of the symbolic debugging records and also supports the -SYM option in order to suppress the display of various debugging records. (For details on the -SYM option refer to the Linker above. The default is FULL.)

DumpCode

- Dumpcode output has been improved cosmetically.

Converting Between pre-3.0 Str255's and 3.0 Str255's

Tom Taylor
16 Mar 88

The definition of Str255 (and all of its derivatives like Str63, Str27, etc.) was changed in MPW 3.0. The pre-3.0 definition was a structure with two elements, a byte length, and an array of text. The new definition of Str255 is:

```
typedef unsigned char Str255[256];
```

The new definition is not backward compatible with the old version. If you use Str255's in your code or use structures that have Str255's (or derivatives) imbedded in them (such as SFReply's), you will have to make source code changes.

Here are a few examples:

Pre-3.0 Source	*changes to*	3.0 Source
`Str255    string;`		`Str255    string;`
`string.length`		`*string` or `string[0]`
`string.text`		`&string[1]`
`&string`		`string`
`&string.length`		`string`

MPW 3.0a2 Interface Release Notes:
24 June 1988

These release notes reflect the differances between MPW 3.0a1 and MPW 3.0a2.

PInterfaces:

The major news in PInterfaces is that we have made the individual little files able to be included or used in any order now.

The unfortunate side affect of this is that the new Pascal compiler must be used with these interfaces. You cannot use a previous version of Pascal when trying to isolate bugs introduced either by the compiler output or new interfaces.

```
AppleTalk.p:
        In routines: DDPOpenSocket, DDPCloseSocket, ATPOpenSocket & ATPCloseSocket
    SignedByte's changed to Byte.
DiskInit.p:
        Type HFSDefaults corrected to be unpacked record.
        The function: DIBadMount was corrected to return an OSErr. However, it
    can in fact return other values as well, so it will be changed back to
    integer for MPW 3.0.
Errors.p:
        Added constant: nmTypErr.
        Removed constants: iTabPurgErr, noColMatch, qAllocErr, tblAllocErr,
    overRun, noRoomErr, seOutOfRange, seProtErr, i2CRangeErr, gdBadDev,
    reRangeErr, seInvRequest, seNoMemErr & smFHBlkDispErr.
Files.p:
        Added to cases to HParamBlockRec: AccessParam, ObjParam, CopyParam &
    WDParam.
        Moved to OSUtils.p: ParamBlkType, QTypes, FInfo, VCB, DrvQEl, ParamBlockRec,
    VBLTask, EvQEl, QHdr & QElem.
        Added constants & types for SetVersion: developStage, alphaStage,
    betaStage & finalStage; & NumVersion & VersRec.
Lists.p:
        ListRec.cellArray corrected to be ARRAY [1..1] OF INTEGER;
Memory.p:
        Function SystemZone changed to be inline.
        Added routine: MFTopMem.
Menus.p:
        Added array: MCTable.
        SetItemIcon & GetItemIcon corrected to use Byte.
OSEvents.p:
        The definition of PPostEvent in "Inside Macintosh" was wrong. The
    value returned by the ROM routine for qEl is a pointer, now EvQElPtr.
OSIntf.p:
        Added include of Sound.p:
OSUtils.p:
        Added constant: curSysEnvVers.
        Moved from Files.p: ParamBlkType, QTypes, FInfo, VCB, DrvQEl, ParamBlockRec,
    VBLTask, EvQEl, QHdr & QElem.
        Added new calls: SetCurrentA5 & SetA5.
Packages.p:
        Added constants: zeroCycle, myd, dym, ydm, longDay, longWeek, longMonth,
    longYear, supDay, supWeek, supMonth, supYear, verIreland, verKorea, verChina,
    verTaiwan, verThailand, minCountry & maxCountry.
Palettes.p:
        Added calls: CopyPalette & NSetPalette.
        Added constants for NSetPalette: pmNoUpdates, pmBkUpdates, pmFgUpdates
```

& pmAllUpdates
ColorInfo.ciFlags & ColorInfo.ciPrivate fields have been combined into ColorInfo.ciDataFields. This will change in "Inside Macintosh".
Palette record has been changed to agree with "Inside Macintosh".

Picker.p:
GetColor: removed VAR from prompt parameter.

Printing.p:
TLong varient record finished.

Quickdraw.p:
Added arithmatic transfer mode constants: blend, addPin, addOver, subPin, addMax, subOver, adMin & transparent.
Added constant: pHiliteBit.
Type ColorTable corrected to be unpacked record.
Routine ScalePt corrected to declare pt as a VAR.
Routine CalcCMask corrected inline code.
Added definition for: CWindowPtr.

ROMDefs.p:
Removed constant: drHwGM.
Added constant: date.

Script.p:
Added constants: smCharLeft & smCharRight.

SCSI.p:
Partition.pmProcessor corrected to be array of CHAR.
SCSIComplete wait parameter corrected to be: LONGINT.

Sound.p:
Corrected value of constant: noteSynth.
Removed constant: midiSynth.
Added constants: midiSynthIn, midiSynthOut, midiInitChanFilter & midiInitRawMode. Notice constants start with lower case "midi" rather than upper case as in "Inside Macintosh".

TextEdit.p:
STElement corrected to be unpacked record.

ToolUtils.p:
ScreenRes parameter scrnVRes corrected to be VAR.

Traps.p:
Corrected _PurgeSpace trap number.

Types.p::
Added type: Str31.

Windows.p:
Added types: CWindowRecord & CWindowPeek.
Routines NewCWindow & GetNewCWindow were changed to use CWindowPtr's.
THIS WILL BE CHANGED BACK TO REGULAR WindowPtr's AT THE NEXT RELEASE!

CIncludes:

The flag for use by C++ has been changed from our original "c_plusplus" to the official "__cplusplus". Also, since C++ may construct new names for functions when compiling, functions are now bounded by a "__safe_link" flag to keep this from happening in the Macintosh interfaces.

With the change of string definitions from structs to character arrays, there were many parameters which still had *Str255. These have now been corrected to Str255 only.

Desk.h:
Added constant: goodbye.

Devices.h:
Control call corrected: * removed from csParamPtr parameter.

Dialogs.h:
GetAlrtStage & ResetAlrtStage now inline.

DisAsmLookUp.h:
Use void * for C type checking in: validMacsBugSymbol, endOfModule & showMacsBugSymbol.

Errors.h:
        Added constant: nmTypErr.
        Removed constants: iTabPurgErr, noColMatch, qAllocErr, tblAllocErr,
    overRun, noRoomErr, seOutOfRange, seProtErr, i2CRangeErr, gdBadDev,
    reRangeErr, seInvRequest, seNoMemErr & smFHBlkDispErr.
Events.h:
        GetDblTime & GetCaretTime now inline.
Files.h:
        Added to cases to HParamBlockRec: AccessParam, ObjParam, CopyParam &
    WDParam.
        Moved to OSUtils.p: ParamBlkType, QTypes, FInfo, VCB, DrvQEl, ParamBlockRec,
    VBLTask, EvQEl, QHdr & QElem.
        Added constants & types for SetVersion: developStage, alphaStage,
    betaStage & finalStage; & NumVersion & VersRec.
        In function unmountvol, parameter vRefNum corrected.
Fonts.h:
        Added structs: WidEntry, WidTable, AsscEntry, FontAssoc, StyleTable,
    NameTable, KernPair, KernEntry & KernTable.
        Function getfnum: familyID parameter corrected.
Graf3D.h:
        Function InitGrf3d: port parameter corrected.
Lists.h:
        Added array: DataArray.
Memory.h:
        Functions GetApplLimit, SystemZone, ApplicZone, GZSaveHnd & TopMem
    changed to be inline.
        Added routine: MFTopMem.
OSEvents.h:
        The definition of PPostEvent in "Inside Macintosh" was wrong. The
    value returned by the ROM routine for qEl is a pointer, now EvQElPtr.
OSUtils.h:
        Added constants: curSysEnvVers, sortsBefore, sortsEqual & sortsAfter.
        Functions GetTrapAddress, SetTrapAddress, NGetTrapAddress & NSetTrapAddress
    changed to use longs per "Inside Macintosh", even though ProcPtr is more correct.
        Added new functions: SetCurrentA5 & SetA5.
Packages.h:
        Added constants: zeroCycle, myd, dym, ydm, longDay, longWeek, longMonth,
    longYear, supDay, supWeek, supMonth, supYear, verIreland, verKorea, verChina,
    verTaiwan, verThailand, minCountry & maxCountry.
Palettes.h:
        Added functions: CopyPalette & NSetPalette.
        Added constants for NSetPalette: pmNoUpdates, pmBkUpdates, pmFgUpdates
    & pmAllUpdates
        ColorInfo.ciFlags & ColorInfo.ciPrivate fields have been combined into
    ColorInfo.ciDataFields. This will change in "Inside Macintosh".
        Palette record has been changed to agree with "Inside Macintosh".
Picker.h:
        Corrected functions HSV2RGB & RGB2HSV to use HSVColor
Printing.h:
        Added constants: iPrPgFst, iPrPgMax, iPrRelease, lPrLFStd, iFMgrCtl,
    iMemFullErr, iIOAbort, pPrGlobals, bUser1Loop, bUser2Loop, lHiScreenBits,
    lHiPaintBits, iPrEvtCtl, lPrEvtAll & lPrEvtTop.
        Added structs: TPrVars, TPfHeader & TPrDlg.
        Corrected struct from TSetRslBk to TSetRslBlk
        Capitalization corrected on fields: TGetRslBlk.xRslRg & TGetRslBlk.yRslRg.
        Field corrected from TDftBitsBlk.iE to TDftBitsBlk.iError.
PrintTraps.h:
        Added constants: iPrPgFst, iPrPgMax, iPrRelease, pPrGlobals, bUser1Loop,
    bUser2Loop, fNewRunBit, fHiResOK, fWeOpenedRF, lPrLFStd, iFMgrCtl, iMscCtl,
    iPvtCtl, iMemFullErr, lHiScreenBits, lHiPaintBits, iPrEvtCtl, lPrEvtAll & lPrEvtTop.
        Added structs: TPrVars, TPfHeader & TPfPgDir.
        Capitalization corrected on fields: TGetRslBlk.xRslRg & TGetRslBlk.yRslRg.
        Corrected struct from TSetRslBk to TSetRslBlk

Field corrected from TDftBitsBlk.iE to TDftBitsBlk.iError.

Quickdraw.h:

Added arithmatic transfer mode constants: blend, addPin, addOver, subPin, addMax, subOver, adMin & transparent.

Added constant: pHiliteBit.

Region.rgnData & Picture.picData fields removed to match "Inside Macintosh".

CGrafPort.rgnSave & CGrafPort.polySave changed to Handle to agree with "Inside Macintosh".

Added definition for: CWindowPtr.

Routine ScalePt corrected to declare pt as a VAR.

Routine CalcCMask corrected inline code.

Routine InitGraf globalPtr parameter now void * for C type checking.

ROMDefs.h:

Added constant: date.

Script.h:

Added constants: smCharLeft, smCharRight & emCurVersion.

SCSI.h:

Field Block0.pad renamed Block0.ddPad.

SCSIComplete wait parameter corrected to be: long.

SegLoad.h:

UnloadSeg now unloads void * for C typechecking.

Serial.h:

Added constants: ainRefNum, aoutRefNum, binRefNum & boutRefNum.

Slots.h:

FHeaderRec corrected to be struct.

Sound.h:

Corrected value of constant: noteSynth.

Removed constant: midiSynth.

Added constants: midiSynthIn, midiSynthOut, midiInitChanFilter & midiInitRawMode. Notice constants start with lower case "midi" rather than upper case as in "Inside Macintosh".

SoundHeader corrected to be struct.

Capitalization corrected on struct: ModRef.

Added function: SndAddModifier.

StdDef.h:

Changed definition of NULL from 0L to just 0 for C++.

SysEqu.h:

Removed DeskCPat & DeskPatDisable.

Added ABusVars & ABusDCE.

ToolUtils.h:

ScreenRes parameter scrnVRes corrected to be VAR.

Traps.h:

Corrected capitalization of _Dequeue & _Enqueue.

Added _LoadScrap & _UnloadScrap (in addition to _LodeScrap & _UnlodeScrap).

Corrected _PurgeSpace trap number.

Added _NMInstall & _NMRemove.

Types.h:

Changed definition of NULL from 0L to just 0 for C++.

Added type: Str31.

Windows.h:

Added types: CWindowRecord & CWindowPeek.

Fields AuxWinRec.awFlags & AuxWinRec.awResrv corrected to right type.

Routines NewCWindow, newcwindow & GetNewCWindow were changed to use CWindowPtr's. THIS WILL BE CHANGED BACK TO REGULAR WindowPtr's AT THE NEXT RELEASE!

AIncludes:

ATalkEqu.a:

Moved to SysEqu.a:: AbusVars & AbusDCE.

PackMacs.a:

Added constants: zeroCycle, myd, dym, ydm, longDay, longWeek, longMonth, longYear, supDay, supWeek, supMonth & supYear.

PrEqu.a:
        Updated values of: lPrEvtAll & lPrEvtTop.

QuickEqu.a:
        Added arithmatic transfer mode constants: blend, addPin, addOver, subPin, addMax, subOver, adMin & transparent.

ROMEqu.a:
        Removed constant: drHwGM.

SysEqu.a:
        Moved from ATalkEqu.a:: AbusVars & AbusDCE.

SysErr.a:
        Added constant: nmTypErr.
        Removed constants: iTabPurgErr, noColMatch, qAllocErr, tblAllocErr, overRun, noRoomErr, seOutOfRange, seProtErr, i2CRangeErr, gdBadDev, reRangeErr, seInvRequest & seNoMemErr.

TimeEqu.a:
        Constant changed: msQSize.

Traps.a:
        Added new traps: _NMInstall & _NMRemove.

Interface.o:

Removed glue for SystemZone, GetApplLimit, ApplicZone, GZSaveHnd, TopMem, GetAlrtStage, ResetAlrtStage, GetDblTime & GetCaretTime which are now provided inline.

CInterface.o:

The function rename was changed to fsrename to avoid ANSI conflict.

MPW 3.0a1 INTERFACE RELEASE NOTES

20 April 1988

by Keithen Hayenga

INTRODUCTION:

Interfaces and libraries to the Macintosh system and toolbox calls for MPW 3.0a1 were changed in 3 major ways: The Pascal interfaces were broken up into many smaller interface files corresponding to "Inside Macintosh". The C header files, CIncludes, had major syntax changes made possible or necessitated by new C and C++ compilers. Reflecting the changes in the C compilers and the C header files, CInterface.o became much smaller.

PINTERFACES:

A major proposal for the MPW 3.0 interfaces is that all constants, record definitions, and function declarations in assembler, c, and Pascal will be organized by manager as defined in Inside Macintosh. This will make it easier to find documentation for any interfaces or, conversely, know where the declarations can be found for any documented calls. It will also improve the maintainability and accuracy of the interfaces if the same information can be found in corresponding files for all target languages. The main impact of this reorganization has been to Pascal. Pascal interfaces now consist of many smaller files which match the file names found in CIncludes rather than the ubiquitous OSIntf.p and ToolIntf.p:

```
AppleTalk.p     Controls.p      Desk.p          DeskBus.p
Devices.p       Dialogs.p       DiskInit.p      Disks.p
Events.p        Files.p         Fonts.p         Graf3D.p
Lists.p         Memory.p        Menus.p         OSEvents.p
OSUtils.p       Packages.p      Palettes.p      Picker.p
Printing.p      PrintTraps.p    Quickdraw.p     Resources.p
Retrace.p       ROMDefs.p       Scrap.p         Script.p
SCSI.p          SegLoad.p       Serial.p        ShutDown.p
Slots.p         Sound.p         Start.p         TextEdit.p
Timer.p         ToolUtils.p     Types.p         Video.p
Windows.p
```

In order to not break existing code, all pre 3.0 PInterface files not listed above must also be available. The files listed below are units of the same name which include the proper smaller files listed above.

```
MacPrint.p      MemTypes.p      OSIntf.p        PackIntf.p
PaletteMgr.p    PickerIntf.p    SCSIIntf.p      ToolIntf.p
VideoIntf.p
```

In addition, error numbers which used to be distributed throughout the Pascal interfaces are now collected in Errors.p. Also added is a Traps.p file, which is useful to check for unimplemented traps.

CINCLUDES:

There were two major changes to the contents of the CInclude files:
1. The structure of pascal-style strings has been redefined to be an array of unsigned char instead of a structure with a length byte. 2. The upper/lower case spelling conventions for functions using strings and points has been changed from MPW 2.0 to MPW 3.0. TO CONVERT SOURCE FILES TO USE THE NEW CIncludes, USE THE CCvt script utility!

All callable functions are now declared within the CIncludes files. Those which work as described in "Inside Macintosh" are now spelled exactly the same as in the book. They are declared to be of type pascal, which allows the use of multiple inline syntax or linkage to the all upper-case glue found in Interface.o. This produces more efficient, easier to maintain code. Those functions which use Points, Cells, or strings (including Str255's) exist in two varieties: Those which pass the Points and Cells by value and accept Pascal style strings were introduced in MPW 2.0 with all upper-case spellings. Since these correspond to the method of accessing the ROMs described in "Inside Macintosh", they are now spelled with the same mixed case. Those calls which were formerly spelled with mixed case passed Points and Cells by address and required glue to convert null terminated C strings to Pascal strings and back. These calls are now spelled with the more "C-like", all lower case spelling and require glue found in CInterface.o. The actual letters making up the names of these routines have remained the same as before with only one exception: Rename. The routine which passes a C string to the Macintosh file system call would collide with the standard C rename function if it were spelled the same with only case change. To avoid this, this version of the call to the Mac file system has been renamed: fsrename.

Another big change for CIncludes is that type checking has come to the C world in the form of C++. In order to allow the use of string literals as parameters for calls that took Str255's, it was necessary to redefine it from a struct with seperate length byte to an array of unsigned char! THIS WILL PROBABLY REQUIRE CHANGES TO YOUR SOURCES FILES. See the paper titled "Str255" for more information.

The actual files contained within the CIncludes folder has also changed:

With the addition of the ANSI C specified Time.h header file, the Macintosh Time Manager header file was renamed Timer.h.

The Palette Manager contains a Palette structure. In the Pascal world, we could not have units and records with the same name, so the file was renamed Palettes.p. For consistancy, the corresponding file in C, Palette.h, has also been renamed Palettes.h. A Palette.h file still remains in CIncludes, but now simply includes Palettes.h.

The much requested low memory equates have finally been added to the CIncludes. They are in a new file: SysEqu.h.

AINCLUDES:

There are no major changes in the AIncludes for MPW 3.0 alpha.

INTERFACE.O:

There are no major changes in Interface.o for MPW 3.0 alpha.

CINTERFACE.O:

Since the new C header files contain multiple inline statements which no longer require glue or which link with existing glue in Interface.o, many less calls must be maintained in CInterface.o. This library now contains only the C specific code required to convert null terminated (C stings) to Pascal style or dereference addresses of Points or Cells. All the entry points in CInterface.o are lower case only to match the convention of the CIncludes.

In the event of precompiled object modules which require linking with older, mixed case glue to convert strings or points, it is possible to also link with an older version of CInterface.o. The mixed case calls will not conflict with the all upper case calls of Interface.o or the all lower case calls of the new CInterface.o.

C CONVERSION SCRIPT:

To convert C source code from MPW 2.0 standards for functions using strings or points to the MPW 3.0 conventions, a script, CCvt, is provided. As of MPW 3.0 alpha, CCvt now has a -p, progress, option and accepts full shell expression for file name input. It also uses the finalized canon dictionaries: CCvtUMx.dict and CCvtMxL.dict

#2xx: Setting and Restoring A5

See also: Operating System Utilities
Technical Note #135
Technical Note #180

Written by: Andrew Shebanow 6/15/88

The routines SetupA5 and RestoreA5 do not work properly when used with some optimizing Pascal and C compilers. Two new routines, SetCurrentA5 and SetA5, are provided here which should work with any compiler.

Introduction

The inline glue routines SetupA5 and RestoreA5 are often used by completion routines, VBL tasks and interrupt handlers written in C and Pascal to get access to an application's global variables. (**Note:** please see Tech Note #180 for guidelines on accessing A5-relative data under MultiFinder.) Unfortunately, these routines play fast and loose with the stack pointer.

Newer, more sophisticated, optimizing compilers (for instance, MPW C 3.0) will often leave function parameters on the stack across multiple function calls, removing the arguments for several functions with a single instruction. This significantly reduces code size and execution time, at the expense of a small amount of additional stack usage. As a side effect, this optimization **breaks** the SetupA5 and RestoreA5 glue.

This Tech Note describes a pair of inline glue routines which have more functionality than SetupA5/RestoreA5, without making assumptions about a compiler's stack handling. These routines are provided as a standard part of the MPW Pascal 3.0 and MPW C 3.0 packages, in the files "`osUtil.p`" and "`osUtil.h`", respectively.

The Old Way

The INLINE code for SetupA5 was:

```
    MOVE.L  A5,-(A7)         ; leave old A5 on stack: Danger Will Robinson!
    MOVE.L  CurrentA5,A5     ; set current A5
```

The INLINE code for RestoreA5 was:

```
    MOVE.L  (A7)+,A5         ; pop old A5 off stack
```

The problem here is that SetupA5 leaves the old value of A5 on the stack, and RestoreA5 assumes that the stack pointer is still valid. If the programmer mistakenly calls RestoreA5 within a called subroutine, the value that is popped off the stack and stored in A5 will be garbage. Of course, the "garbage" could be something moderately useful like a return address. Even if the calls are at the same level, the current top of stack cannot be guaranteed with an optimizing compiler. The MPW C compiler, for example, often pushes arguments on the stack, calls a function, pushes more arguments on the stack, calls another function, and then adjusts the stack in one fell swoop.

The New, Totally Cool Way

The solution to this distressing problem is provided by two new functions, SetCurrentA5 and SetA5. Both of these functions return the old value of A5 to the caller as a result, which can be stored off and restored at some time in the not-too-distant future.

The interface for SetCurrentA5, along with its corresponding implementation as a subroutine call, is:

```
FUNCTION SetCurrentA5 : LONGINT;
        INLINE $2E8D, $2A78, $0904;              /* MPW Pascal */
pascal long SetCurrentA5(void) =
        { 0x2E8D, 0x2A78, 0x0904 };              /* MPW C (3.0 only) */

; Subroutine version for those who aren't using a compiler capable
; of handling multiple word INLINE functions:
SetCurrentA5  PROC EXPORT
              MOVE.L  A5,4(A7)       ; store old A5 as function result
              MOVE.L  CurrentA5,A5   ; set A5 to real value
              RTS
              ENDPROC
```

When you want to restore the old value of A5, or just want to change A5 to some other value, you can use the routine:

```
FUNCTION SetA5 (newA5 : LONGINT) : LONGINT;
        INLINE $2F4D, $0004, $2A5F;              /* MPW Pascal */
pascal long SetA5 (long newA5) =
        { 0x2F4D, 0x0004, 0x2A5F };              /* MPW C (3.0 only) */

; Subroutine version for those who aren't using a compiler capable
; of handling multiple word INLINE functions:
SetA5         PROC EXPORT
              MOVE.L  (A7)+,A0       ; save return address
              MOVE.L  A5,4(A7)       ; store old A5 as function result
              MOVE.L  (A7)+,A5       ; set A5 to passed value, pop argument
              JMP.L   (A0)
              ENDPROC
```

Here is a small piece of sample code, written in ANSI C, which demonstrates a typical use of these routines:

```
#include <Types.h>
#include <Files.h>

extern int aGlobal;                /* a global variable */

/*
 * MyCompletionRoutine:
 *
 *     This routine is called by an assembly language stub that
 * takes its parameters out of D0 and A0.
 */

void MyCompletionRoutine(ParmBlkPtr pb, OSErr result)
{
    long oldA5;

    oldA5 = SetCurrentA5();        /* set current a5 */
    aGlobal = -1;                  /* do some work... */
    (void) SetA5(oldA5);           /* restore previous a5 */
}
```

We recommend that you switch over to the new routines as soon as possible, no matter what development system you use.

Macintosh Programmers Workshop 3.0 Shell ERS

Authors: Dan Allen, John Dance, Jeff Parrish, Peter Potrebic
Date: March 7, 1988

Changes in ERS

This section lists changes in the ERS document since the last printing.

- **The syntax for TileWindows and StackWindows has changed**
- **A new shell variable {IgnoreCmdPeriod}**
- **Delimeter selection will not beep when no match is found.**
- **New shell variables {NewWindowRect} and {ZoomWindowRect}.**
- **The TileWindows and StackWindows menu commands provide more options using {TileOptions} and {StackOptions}.**
- **Shift will modify the direction of Find and Find and Replace also.**
- **User defined delimiters (the {Delimiter} variable) will not be supported.**

Introduction

This document is intended to specify all of the enhancements that will be made to the Shell/Editor for MPW 3.0. By necessity, the number of enhancements will be small. Many of the enhancements will be made to support other, more strategic parts of the MPW 3.0 project such as Projector (project/source code control system). Enhancements which would require architectural changes were avoided. For the most part, this release is viewed as a chance to make the Shell functionally complete. Our goal was to simplify difficult or impossible tasks. For example: we will be adding a command to find out the size and location of a window because there is currently no way to find out this information from a script.

The 3.0 Shell will have the same software and hardware requirements as the 2.0 Shell: it will run on any of the machines in the Macintosh family so long as they have at least one megabyte of memory and 128K or larger ROMs with System 4.1 or later.

For the remainder of this document, the word "shell" will refer to the half of the MPW Shell which provides the command interpreter for scripts, runtime memory management, resource caching, etc. The word "editor" will, as you might guess, refer to the half of the MPW Shell which provides all the editing services which you have probably come to know and cherish. This document applies to both the shell, and editor halves of the MPW 3.0 Shell.

Items marked by a "→" are enhancements that we would really like to include but may not be able to due to time constraints.

Please feel free to direct your comments to any member of the Shell team: Dan Allen (x2801), John Dance (x2232), Jeff Parrish (x2395), or Peter Potrebic (x6494).

Enhancements to the Shell

Enhancements, other than performance improvements, that will be made to the shell fall roughly into one of two categories: additions to existing commands, and support for MultiFinder™ and Projector.

Additions to Existing Commands

- The Date command will have more output formats. For more details, see the attached manual pages.
- Stdout and stderr will be directed to the same place with the Σ (option-w) character. "The summation of all output..." The syntax will be as follows:

Σ *name*	Standard output and diagnostic output replace the contents of *name*. File *name* is created if it doesn't exist.
ΣΣ name	Standard output and diagnostic output are appended to *name*. File *name* is created if it doesn't exist.

- A "directory path" variable (similar to the {Commands} variable) for changing current directories will be added. See the Directory manual page for more information.
- Numeric variables will be added to the Shell command language. The manual page for the Evaluate command describes numeric variables in detail.
- Evaluate will be enhanced to allow output in different radices. See the Evaluate manual page for more information.

Support for MultiFinder™ and Projector

- SpinCursor will be modified to allow background operation of MPW.
- Open projector windows will be listed at the end of the Window menu. The projector windows will be separated from the normal windows by a disabled line. Selecting a projector window in the list will bring that window to the front.

Enhancements to the Editor

Enhancements to the editor will be made to: add new commands, extend existing commands, remove existing restrictions, extend the user interface, and support MultiFinder™ and Projector.

New or Extended Commands

- A -q (quiet) option will be added to the Request command.This option allows the Request commands to not return an error message, thus allowing a script to continue its execution regardless of user input. See the manual page for more details.

- A -c (cancel) option will be added to the Close command. This option allows the user to cancel a close operation. (Normally this would only be useful from a script when the user wanted to close all files that had not been modified.) See the manual page for more details.

- The faccess interface will be extended to allow Tool access to the current selection and window size and position. The routine faccess will be modified to allow the examination and modification of the current selection, the top of window position within a file, and the window size and position.

  The current selection is described by a pair of long integer values for the starting and ending position and another long value for the character that will be displayed on the first line of the window. These positions are offsets from the beginning of the file, with the first position in the file being 0. This information is communicated with faccess in the following format:

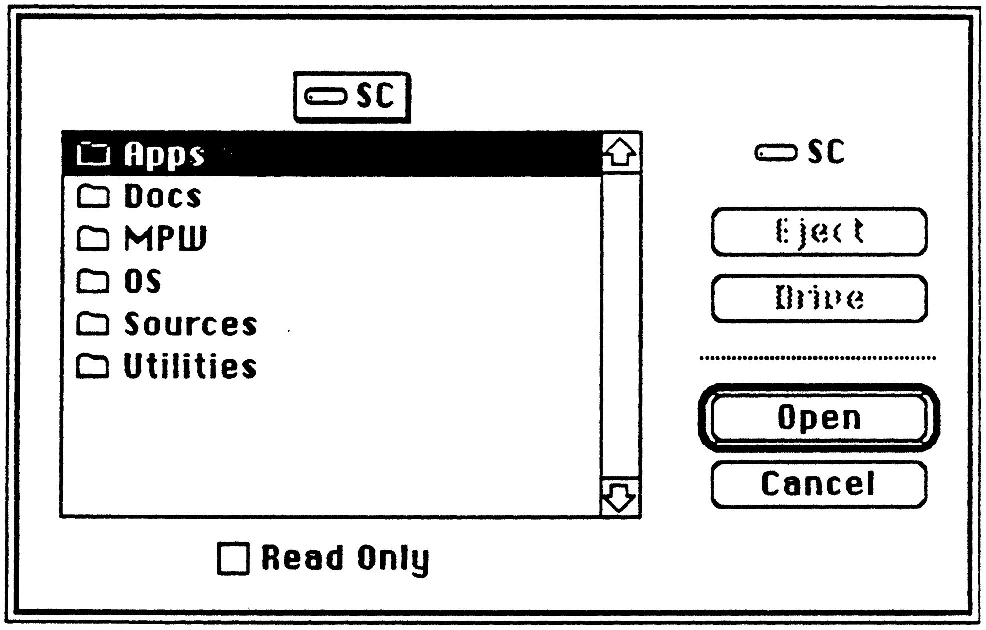
  ```
  struct SelectionRecord {
          long startingPos;
          long endingPos;
          long dispayTop
  };
  ```

  The current window size and position is a rectangle in global coordinates.

  The new commands to faccess will be: F_GSELINFO, F_SSELINFO, F_GWININFO, and F_SWININFO. F_GSELINFO gets the selection information for the MPW text file filename, arg is a pointer to a selection record. F_SSELINFO sets the selection information for the MPW text file filename, arg is a pointer to a selection record. Faccess works on both windows and files. The display will start on the line that contains the character displayTop. DisplayTop does not have to be the first character in a line. The window will not automatically scroll horizontally to display the actual character specified. It is invalid to set startingPos less than zero, greater than endingPos or greater than the length of the file. It is also invalid to set displayTop to a value greater than the length of the file. If displayTop is negative, it will be ignored, and only startingPos and endingPos will be used. If any invalid positions are specified with F_SSELINFO, faccess returns -1, and the global errno will contain the "invalid parameter" code. F_GWININFO gets the current window position. The parameter arg is a pointer to a rectangle to store the information. The rectangle is in global coordinates. F_SWININFO sets the current window position, arg is a pointer to a rectangle specifying the new size and position. If the window size is invalid, or the rectangle is completely off the screen, faccess returns -1 and the global errno will contain the "invalid parameter" code. (For more information about faccess see Chapter 3, The Standard C Library in The MPW C Reference Manual.) Note: This change requires changing the MPW C, Pascal, and ASM Reference Manuals, and the file control interface files (FCntl.h, IntEnv.p, IntEnv.a) in CIncludes, PInterfaces, and AIncludes directory.

• The direction of the Find Same, Find Selection, and Replace Same commands may be modified by holding down the shift key when the menu or command key is selected. This modifies the direction for the current command only. It also does not have any effect on the shell search variables. This will allow the user to easily search both forward and backwards through a file. The direction of Find and Find and Replace may be modified by holding down the shift key when the dialog is closed with the Find, Replace, or Replace All buttons.

• A read-only check box will be added to the "Open..." dialog. This enhancement will allow a file to be opened read-only interactively through the StdFile GetFile dialog box. The check box will be placed under the cluster of buttons already in GetFile. The dialog will look something like this:



- You will be able to inquire a window's size, etc.from a script. The MoveWindow and SizeWindow commands will be extended to provide this information. If MoveWindow and SizeWindow are entered without any coordinates, then they return the upper-left corner position and the window's size respectively. For more details see the attached new manual pages.

- TileWindows and StackWindows will be improved to make them more useful by allowing more options for the layering of windows. See the attached new manual pages for details. The TileWindows and StackWindows menu commands may be used to tile or stack windows. The variables {TileOptions} and {StackOptions} are used by the TileWindows and StackWindows menu commands. Initially, the two variables are undefined. If the option key is held down while selecting either menu item, the worksheet will be included in the tile or stacking operation. (You may always include the worksheet in tiling or stacking without holding the option key by putting "-i" in the {TileOptions} and {StackOptions} variables.)

- Two new variables will help with windows on large screens. There variables are {NewWindowRect} and {ZoomWindowRect}. {NewWindowRect} will be used as the window rectangle when a new window is created. {ZoomWindowRect} will be used when a window is zoomed to "full screen". Both of these rectangles must be visible. If the rectangles specified are not totally visible they will be ignored, and the default rectangles will be used. The coordiates (0,0) are located at the left side of the screen at the bottom of the menu bar. The format of both rectangles are given in top,left,bottom,right order. For example, to create all new windows in the top left corner of the screen 400 pixels wide and 200 pixels high, use the command: Set NewWindowRect 0,0,200,400

- There will be a method to rotate between windows (i.e. send the top window to the bottom). A built-in command will be implemented that rotates between the active MPW windows. For more details see the attached manual pages.

- A Format command will be added to give scripts access to all the features of the Format dialog. This will provide access to the current font, font size, tab size, and the auto indentation and invisible options. For more information see the attached manual pages. **Note**: The Tab and Font commands will be removed. Any scripts using these commands will have to be changed.

- New variables will be created to support user defined document defaults. The variables will allow all formatting options (except display invisibles) to be pre-defined by the user. There variables will be {Font}, {FontSize}, {Tab}, and {AutoIndent}. (The variable {Tab} is already in MPW.) While the Format command and the format menu change the format of a open window. These variables provide default settings for new windows. The definitions are as follows:

{Font}	Specifies the font for a new window.
{FontSize}	Specifies the font size for a new window.
{Tab}	Specifies the tab size for a new window.
{AutoIndent}	Specifies the setting for automatic indenting. (If greater than 0, automatic indenting will be done.)

The pre-defined values are:

{Font}	'Monaco'
{FontSize}	9
{Tab}	4
{AutoIndent}	1

- The format of a locked file can not change. When a locked file is opened, it will be displayed in the default format.

- Three new variables will be pre-defined to provide access to the current search options. (The variable {CaseSensitive} is already defined.) These are:

{SearchWrap}	If greater than 0, searching will wrap around.
{SearchBackward}	If greater than 0, searching will go backward.
{SearchType}	0 = literal, 1 = word, 2 = regular expression

The values of these variables will be predefined as follows:

{CaseSensitive}	0
{SearchWrap}	0
{SearchBackward}	0
{SearchType}	0

- You may now set up scripts to ignore command period. This is useful for critical sections of a script. Command period may be ignored by setting the variable {IgnoreCmdPeriod}. If this variable is defined to a nonzero number, command period will be ignored. {IgnoreCmdPeriod} has a scope just like all other variables, so if you want to ignore command period in inner scopes, you must export the variable. Inner scripts can then allow command period by using Unset. Tools run in the scope that has {IgnoreCmdPeriod} defined will also ignore command period. This overrides any signal handler defined in the tool itself. {IgnoreCmdPeriod} is undefined at startup. Use it with caution.

- The current line number, beginning of selection, and end of selection will be made available by means of the new "Position" command. For details, please refer to the accompanying manual pages in part two of this document.

Remove Existing Restrictions

- The limit on Tab size will be increased. Currently the Editor has an upper limit on the size of Tabs of 20 characters. It is proposed that this limit be increased to 100 characters.

- The allowable length of lines in the Editor will be increased. Currently the Editor has a line length limit of 132 characters: It is not possible to scroll past the 132nd character in a line reliably. It is proposed that this limit be lengthened to 256 characters.

Extend the User Interface

- Different aspects of selecting text by matching delimeters (i.e.: {}, (), []} will change. Rather than selecting the rest of the document when a matching character is not found, the delimeter at the position of the double-click will be selected. During the search, the user may abort by pressing command-period.

- Option–Enter will invoke Commando.

- The amount scrolled horizontally per click will be Increased.

- Opening windows on different screen sizes will be handled better. If a window is in the zoomed out state, it will open to the size of the current screen. Therefore, a zoomed out window will be zoomed out on both a Mac SE and Mac II.

- A new and improved "About Box" will be created.

Part II—Manual Pages

Close—close specified windows

Syntax Close [-y | -n | -c] [-a |*windows*...]

Description Close the window or windows specified by *windows*. If no window is specified, the target window is closed. If changes to the window have not been saved, a dialog requests confirmation of the Close. In scripts you may use the **-y**, **-n** or **-c** options to avoid this interaction. Use the **-a** option instead of *windows* to close all of the open windows (other than the WorkSheet.)

Input None.

Output None.

Diagnostics Errors are written to diagnostic output.

Status Close may return the following status values:

0	No errors
1	Syntax error
2	Any error such as "Window not found."
4	User specified Cancel

Options

`-a` Close all open Shell windows (except for the WorkSheet, which cannot be closed). This option cannot be specified when any *windows* are specified.

`-n` Answer "No" to any confirmation dialogs, causing all of the specified windows to be closed without saving any changes.

`-y` Answer "Yes" to any confirmation dialogs, causing all of the specified windows to be saved before closing them.

`-c` Answer "Cancel" to any confirmation dialogs, effectively causing any modified windows to be left open.

Examples

```
Close
```

Closes the target window, prompting the user with a confirmation dialog box if needed.

```
Close -a -y
```

Saves and closes all open windows.

```
Close -n Test.a Test.r
```

Closes the windows Test.a and Test.r without saving any of the changes.

See also The "File Menu", in Chapter 3.

Date—write the date and time

Syntax	Date [[-a | -s] [-d | -t] [-c *num*]] | [-n]
Description	Writes the current date and time to standard output in a variety of standard and user specified formats. Date arithmetic is supported with the -n and -c options that work with the number of seconds since January 1, 1904.
Type	Miscellaneous.
Input	None.
Output	The full date is written to standard output.
Diagnostics	Errors are written to diagnostic output.
Status	Date may return the following status values:

0	No error.
1	Syntax error.

Options

-a	Abbreviated date. Three-character abbreviations are used for the month and day of the week. For example, Wed, Aug 26, 1987 12:36:48 PM.
-d	Write the date only.
-s	Short date form.. Numeric values are used for the date. The day of the week is not given. For example, 8/26/87 12:36:48 PM.
-t	Write the time only.

`-n`	Returns a numeric value of the current date and time as the number of seconds since midnight on January 1st, 1904.	
`-c`	*num*	Write the date corresponding to *num*, which is interpreted as the number of seconds since midnight on January 1st 1904. The other output format options may be used with this to specify the output format.

Directory—set or write the default directory

Syntax	Directory [-q | *directory*]
Description	If specified, *directory* becomes the new default directory; otherwise the pathname of the current default directory is written to standard output. If *directory* is a leafname, the command searches for *directory* in the directories listed in the Shell variable {DirectoryPath}. If the variable is undefined then the command looks in the current directory. *Note:* To display a directory's contents, use the Files command.
Input	None.
Output	If no directory is specified, the default directory pathname is written to standard output.
Diagnostics	Errors are written to diagnostic output.
Status	Status code 0 is returned if the command succeeded; otherwise 1 is returned.

Option

`-q`	Don't quote the pathname that is written to standard output. (Normally, a directory name is quoted if it contains spaces or other special characters.)

Examples

```
Directory
```

Write the pathname of the current directory to standard output.

```
Directory HD:MPW:AExamples:
```

Set the default directory to the folder AExamples in the folder MPW on the volume HD. The final colon is optional.

```
Directory Reports:
```

Set the default directory to the volume Reports. (Note that volume names must end in a colon.)

```
Directory :Include:Pascal:
```

Set the default directory to the folder Pascal in the folder Include in the current default directory.

```
Set DirectoryPath ":,{MPW},{MPW}Projects:"
Directory Tools
```

Set the directory to the Tools directory. The current directory is searched first, followed by the {MPW} directory, and finally the {MPW}Projects directory.

See also "File and Window Names" in Chapter 3
Files and NewFolder commands

Evaluate—Evaluate an expression

Syntax

Evaluate [-h | -o | -b] [*word...*]
Evaluate *name* [*binary operator*]= [*words...*]

Description

The list of words is taken as an expression. After evaluation, the result is written to standard output. Missing or null parameters are taken as zero. You should quote string operands that contain blanks or any of the characters listed in the table below.

The operators and precedence are mostly those of the C language; they're described below.

As an extention to this command assignments can be made to variables. The second form of the evaluate command evaluates the list of words and assigns the result to the variable *name*. The result of the expression is not written to standard output in this case. C sytle operations of the form "+=", "-=", etc. are supported. If *name* is undefined at the time of execution it is interpreted as zero.

Different radices may be used in the input expression, and the result may be output in a different radix by using the **-h**, **-o**, or **-b** options. The default radix is decimal.

Expressions: An expression can include any of the following operators. (In some cases, two or three different symbols can be used for the same operation.) The operators are listed in order of precedence—within each group, operators have the same precedence.

	Operator			**Operation**
1.	(*expr*)			Parentheses are used to group expressions.
2.	-			Unary negation
	~			Bitwise negation
	!	NOT	¬	Logical NOT
3.	*			Multiplication
	÷	DIV		Division
	%	MOD		Modulus division
4.	+			Addition
	-			Subtraction
5.	<<			Shift left
	>>			Shift right
6.	<			Less than
	<=	≤		Less than or equal
	>			Greater than
	>=	≥		Greater than or equal
7.	==			Equal
	!=	<>	≠	Not equal
	=~			Equal—regular expression
	!~			Not equal—regular expression
8.	&			Bitwise AND

9.	^		Bitwise XOR
10.	|		Bitwise OR
11.	&&	AND	Logical AND
12.	||	OR	Logical OR

All operators group from left to right. Parentheses can be used to override the operator precedence. Null or missing operands are interpreted as zero. The result of an expression is always a string representing a number in the specified radix (default decimal).

The logical operators !, NOT, ¬, &&, AND, | |, and OR interpret null and zero operands as false and nonzero operands as true. Relational operators return the value 1 when the relation is true, and the value 0 when the relation is false.

The string operators ==, !=, =~, and !~ compare their operands as strings. All others operate on numbers. Numbers may be either decimal, hexadecimal, octal, or binary integers representable by a 32-bit signed value. Hexadecimal numbers begin with either $ or **0x**, octal numbers begin with a **0** (leading zero), binary numbers begin with a **0b**. Every expression is computed as a 32-bit signed value. Overflows are ignored.

The pattern-matching operators =~ and !~ are like == and != except that the right-hand side is a regular expression which is matched against the left-hand operand. Regular expressions must be enclosed within the regular expression delimiters /…/. Regular expressions are summarized in Appendix B.

Note: There is one difference between using regular expressions after =~ and !~ and using them in editing commands—when evaluating an expression that contains the tagging operator, ®, the Shell creates variables of the form {®*n*}, containing the matched substrings for each ® operator. (See the examples below.)

Filename generation, conditional execution, pipe specifications, and input/output specifications are disabled within expressions, to allow the use of many special characters that would otherwise have to be quoted.

Expressions are also used in the If, Else, Break, Continue, and Exit commands.

Input None.

Output The result of the expression is written to standard output. Logical operators return the values 0 (false) and 1 (true).

Note: To redirect Evaluate's output (or diagnostic output), you'll need to enclose the Evaluate command in parentheses; otherwise, the > or ≥ symbols will be interpreted as expression operators, and an error will occur. (See the third example below.)

Diagnostics Errors are written to diagnostic output.

Status

Format may return the following status values:

0	Valid expression
1	Invalid expression

Options

`-h`	Output the result in hexidecimal. The number will be prefixed with a 0x.
`-o`	Output the result in octal. The number will be prefixed with a 0.
`-b`	Output the result in binary. The number will be prefixed with a 0b.

Examples

```
Evaluate (1+2) * (3+4)
```

Do the computation and write the result to standard output.

```
Evaluate -h 8 + 8
```

Do the computation and write the result to standard output in hexidecimal (0x10).

```
Evaluate lines += 1                # new way
```

The Evaluate command increments the value of the Shell variable {lines} by 1. If {Lines} was undefined before executing the command, {Lines} would be 1 after execution.

```
( Evaluate "{aPathname}" =~
/(([¬:]+:)*)®1≈/ ) > Dev:Null Echo {®1}
```

These commands examine a pathname contained in the variable {aPathname}, and return the directory prefix portion of the name. In this case, Evaluate is used for its side effect of enabling regular expression processing of a filename pattern. The right-hand side of the expression (/(([¬:]+:)*)®1≈/) is a regular expression that matches everything in a pathname up to the last colon, and remembers it as the Shell variable {®1}. Evaluate's actual output is not of interest, so it's redirected to the bit bucket, Dev:Null. (See "Pseudo-filenames" in Chapter 3.) Note that the use of I/O redirection means that the Evaluate command must be enclosed in parentheses so that the output redirection symbol, >, is not taken as an expression operator.
This is a complex, but useful, example of implementing a "substring" function. For a similar example, see the Rename command.

See also "Structured Commands" in Chapter 5

"Pattern Matching (Using Regular Expressions)" in Chapter 6
and Appendix B

Format—set or view the window format

Syntax

Format [-f fontname] [-s fontsize] [-t tabsize] [-a attributes] [-x formatting] [*window...*]

Description

This is a scriptable form of the format option in the Edit menu. It can be used to set the format of the specified list of windows. If no window is specified, the command operates on the target window. If no options are specified (other than -x), the current format settings are written to standard output.

Note: The Format command (and the Format... menu item) modify the format of an existing window. The format related variables such as {Tab} and {Font} are used to initialize the format of a new window.

Input

None.

Output

If the optional parameters are omitted, or the -x option specified, the current format settings are written to standard output.

Diagnostics

Errors are written to diagnostic output.

Status

Format may return the following status values:

0	No errors
1	Syntax error (error in parameters)
2	All other errors

Options

`-f fontname` Changes the font in the specified windows to fontname.

`-s fontsize` Changes the font size in the specified windows to fontsize.

`-t tabsize` Changes the tab size in the specified windows to tabsize.

`-a attributes` Set or clear the invisible and auto-indent states. Attributes is a string composed of the characters listed below. Attributes that aren't listed remain unchanged.

A	autoindent
I	show invisibles

Uppercase letters set the attribute to on, lowercase turn off the attribute.

`-x formatting` This option is used to specify the output format when the current settings are displayed. This option is ignored if any other option is specified. The parameter formatting is a string composed of the following letters, (in any order) where the order determines the field's position in the output. The values specified will be output separated by spaces.

`f`	font name
`s`	font size
`t`	tab size
`a`	autoindent and show invisibles state

Examples

```
Format -f Monaco -t 8 -a A "{target}"
```

Sets the font , tab size, and autoindent in the target window. The font size and invisible settings are not changed.

```
Format -s 12 MyWindow
```

Changes the font size in MyWindow to 12 point.

```
Format "{Target}"
Format -f Monaco -s 9 -t 8 -a Ai
```

After executing the format statement above, another format command with no options will display current settings. This output format may be selected and executed.

```
Format -x tsf
4 9 Monaco
```

Displays only the values of the specified options. This option would be used for easily retrieving one or two values and assigning them to shell variables for latter use.

See also The "Edit Menu", in Chapter 3. "Variables", in Chapter 5.

MoveWindow—move window to h,v location

Syntax MoveWindow [**h v**] [*window*]

Description Moves the upper-left corner of the specified *window* to the location (**h**,**v**) where **h** and **v** are horizontal and vertical integers. The coordinates (0,0) are located at the left side of the screen just below the menu bar. If the location specified would place the window's title bar off the visible screen, an error will be returned. If no window is specified, the target window (the second window from the top) is assumed. If no location is specified, the specified window's location is returned without any effect on the window.

Type Miscellaneous.

Input None.

Output If the **h** and **v** pair is not supplied MoveWindow returns the current location of the specified window.

Diagnostics Errors are written to diagnostic output.

Status MoveWindow may return the following status values:

0	No errors
1	Syntax error (error in parameters)
2	The specified window does not exist.
3	The **h v** location specified is invalid, that is, not on the screen.

Options None.

Examples

```
MoveWindow 72 72
```

Moves the target window's upper-left corner to a point approximately one inch in and down from the upper-left corner of the screen.

```
MoveWindow
```

Returns `MoveWindow 72 72` when executed after the above example.

```
MoveWindow 0 0 "{Worksheet}"
```

Moves the Worksheet window to the upper-left corner of the screen.

See also SizeWindow, ZoomWindow, StackWindows and TileWindows commands.

Position—list position of selection in window

Syntax Position [-c] [-l] [*window*...]

Description Displays the position of the selection in each of the windows specified. If no window is specified, the position of the selection in the Target window is given. By default, the position is displayed as both the line number of the start of the selection and the character positions of the start and end of the selection. The -c option can be used to display only the character positions of the selection. Similarly, the -l option can be used to display only the line number.

Type Miscellaneous.

Input None.

Output The position information is written to standard output.

Diagnostics Syntax errors are written to diagnostic output.

Status Position may return the following status values:

0	No errors.
1	Syntax error.
2	Any other error.

Options

`-l`	Display just the line number of the start of the selection.
`-c`	Display just the character positions of the start and end of the selection.

Examples

```
Position "{Target}" file2
```

Displays the position of the selection in both the Target and file2 in the following form:

```
578 23129,23140
211 8440,8440
```

See also Find command.

Request—request text from a dialog box

Syntax	Request [-q] [-d *default*] [*message*...]
Description	Displays an editable text dialog box with OK and Cancel buttons and the prompt *message*. If the OK button is selected, then all text that the user typed into the dialog box is written to standard output. The **-d** option lets you set a default response to the request.
Type	Miscellaneous.
Input	Reads standard input for the message if no parameters are specified.
Output	Text from the dialog is written to standard output.
Diagnostics	Syntax errors are written to diagnostic output.
Status	Request may return the following status values:

0	The OK button was selected.
1	Syntax error.
4	The Cancel button was selected.

Options

`-d` *`default`*	The editable text field of the dialog box is initialized to *default*.
`-q`	Makes Request quiet, that is, Request always returns a status of either zero or one. This is useful in scripts.

See also Confirm command.

RotateWindows—rotate between windows

Syntax RotateWindows

Description Puts the front MPW window in the back, and brings the second window to the front. Multiple calls to RotateWindows will rotate through all open MPW windows. RotateWindows will only bring MPW windows to the front. (Desk accessory windows will not be rotated.) This command would usually be added to a menu with a command key equivalent. For example: AddMenu 'Extras' 'RotateWindows/®' 'RotateWindows'

Input None.

Output None.

Diagnostics None.

Status RotateWindows returns the following status values:

0	No errors
1	Syntax error (error in parameters)

Options None.

Examples

```
RotateWindows
```

Puts the front MPW window in back, and brings the target MPW window to the front.

See also StackWindows, SizeWindow, MoveWindow and ZoomWindow commands

SizeWindow—set a window's size

Syntax SizeWindow [**h** v] [*window*]

Description Sets the size of the specified *window* to be **h** by v pixels, where **h** and **v** are non-negative integers referring to the horizontal and vertical dimensions respectively. The default *window* is the target (the second window from the top); a specific *window* can optionally be specified. If no size is specified, the current size of the specified window is returned and the window size is not effected.

Type Miscellaneous.

Input None.

Output If no size is specified, SizeWindow returns the current size of the window specified.

Diagnostics Errors are written to diagnostic output.

Status SizeWindow may return the following status values:

0	No errors
1	Syntax error (error in parameters)
2	The specified window does not exist.
3	The **h** v size specified is invalid.

Options None.

Examples

```
SizeWindow 200 200
```

Makes the target window 200 pixels square in size.

```
SizeWindow
```

Returns `SizeWindow 200 200` when executed after the above example.

```
SizeWindow 500 100 "{Worksheet}"
```

Makes the Worksheet window 500 by 100 pixels in size.

See also MoveWindow, ZoomWindow, StackWindows, and TileWindows commands.

StackWindows—arrange windows

Syntax StackWindows [-i] [-h num] [-r t,l,b,r] [-v num] [*windows*...]

Description Automatically sizes and moves all of the specified Shell windows (except the Worksheet) such that they are stacked up slightly staggered and overlapping. If no *windows* are specified, then all open Shell windows are stacked up. Additionally the user may specify the horizontal and vertical staggering constants, otherwise it defaults to 5 pixels horizontally and 20 pixels vertically.

Input None.

Output None.

Diagnostics Errors are written to diagnostic output.

Status StackWindows returns the following status values:

0	No errors
1	Syntax error (error in parameters)

Options

`-i`	Include the Worksheet window when stacking if there is no list of windows specified.
`-h num`	Stack the specified windows with num pixel horizontal spacing. The default is 5 pixels wide. Negative values are not allowed and return a syntax error.
`-r t,l,b,r`	Stack the specified windows within the specified rectangle. The rectangle is specified in top, left, bottom, right order. The default rectangle is the entire screen less the menu bar. The coordinates of the rectangle are separated by commas. If spaces are included, then the rectangle must be inclosed in quotes, such as "10, 10, 500, 300".
`-v num`	Stack the specified windows with num pixel vertical spacing. The default is 20 pixels high—the height of a window's title bar. Negative values are not allowed and return a syntax error.

Examples

```
StackWindows
```

Stacks all of the Shell windows in a neat and orderly fashion.

```
StackWindows -h 10 "{active}" "{target}"
```

Stacks the top two windows (not including the Worksheet) with a vertical spacing of 20 pixels and a horizontal spacing of 10 pixels.

See also TileWindows, SizeWindow, MoveWindow and ZoomWindow commands

TileWindows—arrange windows

Syntax TileWindows [-i] [-h | -v] [-r top,left,bottom,right] [*window*...]

Description Automatically sizes and moves the specified Shell windows (except the Worksheet) such that they are all visible on the screen at once. If no windows are specified then all windows by default are tiled.

Input None.

Output None.

Diagnostics Errors are written to diagnostic output.

Status TileWindows returns the following status values:

0	No errors
1	Syntax error (error in parameters)

Options

`-i`	Include the Worksheet window when tiling if no list of windows is specified.
`-h`	Tile the specified windows in a horizontal fashion allowing the full width of the screen to be used in viewing a file.
`-r` t,l,b,r	Tile the specified windows within the specified rectangle. The rectangle is specified in top, left, bottom, right order. The default rectangle is the entire screen less the menu bar. The coordinates of the rectangle are separated by commas. If spaces are included, then the rectangle must be inclosed in quotes such as "10, 10, 500, 300".
`-v`	Tile the specified windows in a vertical fashion in order to see more lines of a document.

Examples

```
TileWindows
```

Arranges all of the Shell windows in a neat and orderly fashion.

```
TileWindows -h hd:new:main.c hd:new:foo.c
```

Arranges the specified windows as two long horizontal strips.

```
TileWindows -i -v "{active}" "{target}"
```

Arranges the top two windows (including the Worksheet) vertically.

See also StackWindows, SizeWindow, MoveWindow and ZoomWindow commands

Commando's Built-In Editor
and other new features

Starting with version 2.0D3, Commando has a built-in editor that allows you to move and size controls **and edit text labels and help messages**. This feature will hopefully make designing, redesigning, and fine-tuning Commando dialogs much easier. Presently, Commando can only move and size controls. Controls cannot be created, duplicated, or deleted. This means that you still have to manually create the Commando resource, but don't have to worry so much about the coordinates and sizes of the controls. Once the Commando resource has been created, you can simply run Commando, arrange all the controls to your liking, and then simply DeRez the cmdo resource.

Enabling Commando's Editor

In order to enable Commando's built-in editor, the command key must be held down immediately after Commando is launched until the watch cursor appears.

Editing Controls

When Commando is launched with the built-in editor enabled, there are two different Commando modes: 1) normal mode where Commando works just as it always did, and 2) edit mode where controls can be dragged and sized. Holding down the option key puts Commando in edit mode. The option key must be held down to do any selecting, dragging, or growing operation.

Selecting Controls

To select a control, simply press the option key and click on a control. To select multiple controls, press the option and shift keys together and click each control to be selected **or click and drag a marquee around a group of controls**. Clicking a selected control with the shift (and option) key down, unselects that control. Basically, selecting controls works exactly like selecting icons in the Finder with one exception: you must hold down the option key in Commando.

The Commando editor will not allow you to select controls outside of the user control area. For that reason, the coordinates you give when manually creating the Commando resource should fall within the user area.

Moving Controls

Moving controls works as you would probably suspect: simply click and drag a control or selected group of controls. The Commando editor will not allow controls to be dragged outside (actually, no closer than 2 pixels to) the user control area.

Selected controls can be moved one pixel at a time by holding down the option key and pressing the appropriate arrow key.

The top-left corner of the control can be aligned (snapped) to a four pixel grid by holding down the command key while dragging. If a selected group of controls is dragged with the command key down, each of the selected controls' top-left corners will be aligned to the grid.

Sizing Controls

Controls are sized by clicking and dragging the small gray rectangle in the lower left corner of a control. Of course, the option key must be held down to size a control.

Holding down the command key while sizing a control will size the control's height to the recommended Commando height (remember in the Commando documentation that each control has a height that works and looks best). Also, the right edge will be aligned to a four pixel grid. Simply clicking a selected control's grow handle with the option and command keys held down will size the control. List and MultiRegularEntry controls will be sized to the nearest whole line.

Some controls, such as Redirection controls, cannot be resized and have no grow handles.

Hints and Kinks

As mentioned in the Commando documentation, lines and boxes surrounding other controls must be declared later in the Commando resource than the controls they surround. You may encounter situations where you will have to move a control out of the way in order to select a control underneath.

Controls sized and/or moved in nested dialogs do not go back to their original size/position when the nested cancel button is clicked.

Editing Labels

To edit a text title label, simply select it the same way as selecting a control. You can change the text the same way as you would change the text for an icon in the Finder. Once the title is selected, you don't hold down the option key to change the text.

Editing Help Messages

Whenever you select a control, the control's help message gets "locked" in the help window. You can edit the help message like a regular text edit field (note that you don't hold down the option key when editing the help message). The help message stays locked until another control is selected (and then the new control's help is locked) or until all the controls are unselected.

Saving the Modified Commando Dialog

Once any control is sized or moved and the main cancel or do it button is clicked, Commando will prompt you with a save dialog. The save dialog has three options: 1) save the resource, 2) don't save the resource, or 3) cancel the save dialog and go back to Commando for more editing.

When Commando saves the resource, it simply replaces the original resource, wherever it came from. The next time you run Commando on the changed resource, the control positions and sizes will be where you last left them. You can then DeRez the cmdo resource to get the actual coordinates or to simply generate the .r file that will be used in a build.

Version Item in Commando Dialogs

Commando allows you to put a version string in your commando dialogs. The string is centered below the "do it" button. Here is the declaration for VersionDialog:

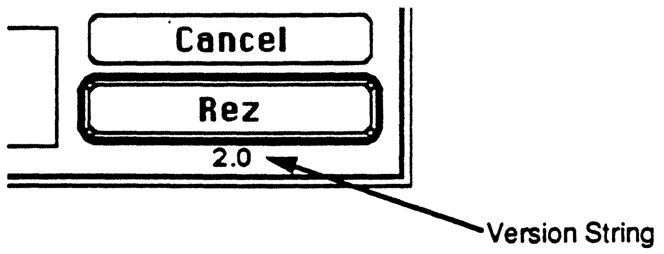
```
case VersionDialog:                          /* Display a dialog when the version # is clicked */
        key byte = VersionDialogID;
        switch {
                case VersionString: /* Version string embedded right here */
                        key byte = 0;
                        cstring;             /* Version string of tool (e.g. V2.0) */

                case VersionResource:        /* Versions string comes from another resource */
                        key byte = 1;
                        literal longint;     /* resource type of pascal string containing version string */
                        integer;             /* resource id of version string */
        };
        cstring;                             /* Version text for help window */
        align word;
        integer         noDialog;/* Rsrc id of 'DLOG' */
                /* NOTE •1: if there is no modal dialog to display when the version
                   string is clicked, set the rsrc id to zero (noDialog).

                 NOTE •2: if the version string comes from another resource (VersionResource),
                 the string must be the first thing in the resource and the string must be
                 a pascal-style string. A 'STR ' resource is an example of a resource that
                 fits the bill.

                 NOTE •3: if the modal dialog is to have a filter proc, the proc
```

```
must be linked as a 'fltr' resource with the same resource id
as the dialog. */
```



The version string may be embedded in the commando resource using the "VersionString" case or the version string may come from a resource using the "VersionResource" case. If the version comes from a resource, the resource must simply contain a rez-style pstring. This can be used with the SetVersion tool to read the 'MPST' resource.

As usual, the help string is a string that is displayed when the version string is clicked. Typically, this help string contains more detailed author/version information.

For extra flair, a dialog may be zoomed out when the version string is clicked. If a dialog is specified, you must give the resource id of the DLOG resource (found in your tool or script) to display. Commando simply calls ModalDialog() with that dialog. If you want to have a custom filter proc, you must compile the filter proc as a standalone resource with a resource type of 'fltr' and with the same id as the DLOG resource. The visible/invisible flag in the DLOG resource should be set to invisible. Commando will move the DLOG window so that the bounds rect specified in the DLOG is relative to the bounds of the Commando dialog.

NOTE: if you do not specify a VersionDialog commando item, Commando will attempt to add one for you by looking for a 'vers' resource with an id of 1. If found, Commando will display the short version string under the "do it" button. When the version string is clicked, Commando will display the long version string in the help window. If a 'vers' (1) resource is not found, Commando will look for a 'vers' (2) resource. If not found, no version string will be displayed.

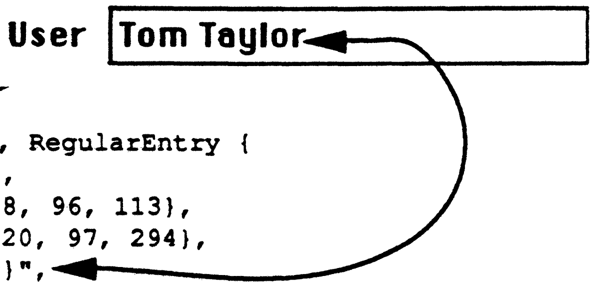
Strings and Shell Variables

It is possible to dynamically change strings in Commando dialogs by having those strings come from shell variables. To make strings come from shell variables, simply make the string like this:

```
"{shell variable}"
```

The string must begin with a '{' and end with a '}'. No leading or trailing spaces are allowed. The shell variable must be an exported variable. If the variable is undefined at the time the Commando dialog is envoked, the variable name with braces will be displayed.

This feature is used in some of Projector's Commando dialogs to display the current user:



```
Or {{-1}}, RegularEntry {
    "User",
    {81, 78, 96, 113},
    {81, 120, 97, 294},
    "{User}",
    ignoreCase,
    "-u",
    "Enter the name of the current user. If no name is"
    " entered, the name in {User} is used."
},
```

Note #1: when Commando is envoked with its built-in editor, shell variable strings will not be expanded to the shell variable values. This is so the strings can be edited and then saved as shell variables and not as the values of shell variables.

Note #2: any string in the Commando resource can be a shell variable. This includes option strings, help strings, titles, etc.

SetVersion—maintain version and revision number

Syntax

Setversion [*option* ...] *file*

Description

Version and revision numbers for an application or MPW tool specified by *file* are assumed to be maintained in the form "*ver.rev*", where *ver* is considered a version number and *rev* a revision number. These values may be displayed by an application's "about box" or when an MPW tool's **-p** option is used. Use SetVersion to independently maintain the version and revision numbers as a resource in the application or tool. Optionally, SetVersion can update a version and revision string in a source file. Pascal, C, and Rez source files are supported.

The current version and revision values are always assumed to be in the specified file's resource fork as a string resource with the resource type 'MPST' and a resource ID of 0 (you can use the **-t** and **-i** options to specify another resource type and ID number if desired). The resource will be created by SetVersion if it is not already there. The string always contains the characters "Version *ver.rev*", where *ver* and *rev* are digits. The version may optionally be prefixed with an arbitrary string (-**prefix**), and the revision may be similarly suffixed with an arbitrary string (-**suffix**) for more complex version numbering (such as "Version x1.23B2").

SetVersion can perform the following functions on the version and revision values:

- Increment the version number by 1 (-**v**).
- Set the version number to a specific value (-**sv**).
- Increment the revision number by 1 (-**r**).
- Set the revision number to a specific value (-**sr**).

The 'MPST' resource attached to the application or tool is considered *the* location of the version and revision. If you attach the 'MPST' resource to the actual application or tool, it will "go" wherever the application or tool goes! Thus the application or tool filename is a required parameter to SetVersion. However, the values contained in the 'MPST' resource can be used to set a corresponding string constant in a source file used to generate the application or tool. This feature is optional, but it should be used for two reasons. First, it explicitly allows the source to reflect the version and revision numbers in the 'MPST' resource. Second, if, for any reason, the 'MPST' resource cannot be accessed, the constant can be used.

The following Pascal code fragment illustrates how the 'MPST' resource and its corresponding source string constant can be used to access the version and revision of an MPW tool. First, in the case of Pascal, the source constant is assumed to be declared as follows (all the formats are discussed under "Options" below):

```
CONST
  Version = '1.2';                          {ver.rev string const.}
```

The following procedure can now be used to get the current version and revision numbers:

```
PROCEDURE GetVerRev(VAR VerRev: Str255);

  VAR
    H: StringHandle;
    i: Integer;

  BEGIN  {GetVerRev - get current "ver.rev"}
    H := StringHandle(GetResource('MPST', 0)); {Get 'MPST' rsrc   }
    IF H = NIL THEN                            {Use string const.}
      VerRev :=  Version                       {if not found      }
    ELSE
      BEGIN
      i := Pos('Version', H^^) + 8;            {Start of ver.rev }
      VerRev := Copy(H^^, i, Length(H^^)-i+1); {Extract from rsrc}
      END;
  END; {GetVerRev}
```

Normally, SetVersion is used with its **-r** option as part of a makefile to automatically increment the revision number each time the application or tool is rebuilt. For each (major) release the version number should be incremented and the revision reset to 0. Note that when SetVersion modifies the application or tool, or updates a source file, the modification date is *not* changed. Therefore, makefiles will not be affected by the use of SetVersion.

Type Tool.

Input The *file* parameter specifies the filename of an application or tool containing the 'MPST' string resource.

Output None.

Diagnostics Errors are written to the diagnostic file.

Status The following status values may be returned:

0	Normal termination
1	Parameter or option error
2	Execution terminated

Options

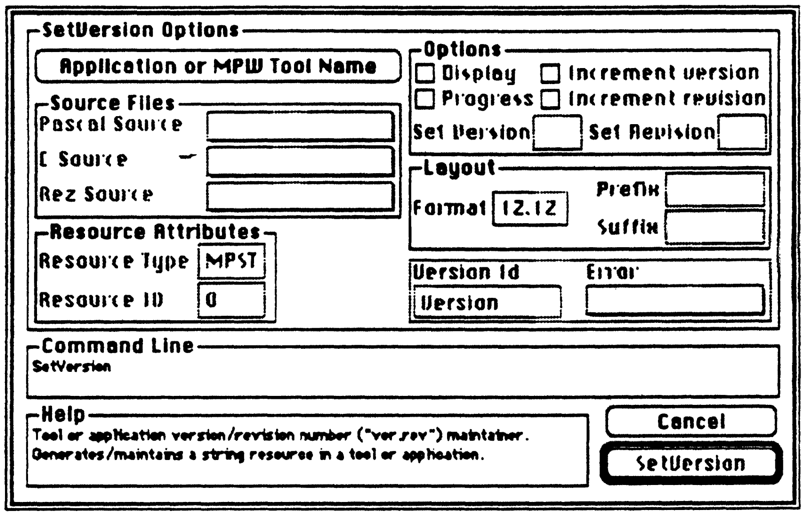
-csource *file* Update the string constant in the C source specified by the *file*. The constant is set to be the same as that specified by the 'MPST' resource string in the application or tool. It is assumed that the constant is defined as a string constant in a #define, somewhere in the first 12800 characters (25 512-byte blocks) of the file, as follows:

```
#defineΔVersion "ver.rev"ΔΔΔΔΔΔΔΔΔΔ/*some comment*/
```

The Δ's indicate required spaces. There may be any number of spaces before the *required* comment. However, because SetVersion edits the line in-place, there must be enough room to allow for changes in the size of the version and revision values—otherwise an error will be reported to the diagnostic file. Case is ignored, and C comments are skipped, when searching for the characters "#defineΔVersion" in the source. The **-verid** may be used to search for a different #define identifier if desired.

-d Write the (updated) version and revision values contained in the 'MPST' resource string to the diagnostic output file.

-fmt *nf.mf* Format the version and revision values according to the specified format. The format of the resource is changed only if the version and/or revision is actually changed (**-sv**, **-v**, **-sr**, **-r**). The format is specified as *nf.mf*, where *f* is either of the letters D or Z, and *n* and *m* are integer values from 1 to 10, which specify the field widths of the version and revision numbers respectively. If the version or revision value is larger than the specified field width, the width is enlarged to contain the entire value. Each field is independently padded up to the specified width with leading zeros or blanks according to the setting of *f*. "D" indicates leading blanks, and "Z" indicates leading zeros. For example, a format of 1Z.3Z for a version/revision value of 10.2 would be formatted as d10.002. The default format is 1Z.1Z. Only the version format (*nf*) or revision format (*.mf*, the period is required) need be specified, allowing the other value to format according to the default.



Resource Type: MPST
Resource ID: 0
Format: 12.12
Version Id: Version

Command Line: SetVersion

Help: Tool or application version/revision number ("ver.rev") maintainer.
Generates/maintains a string resource in a tool or application.

-i *resid* The 'MPST' resource ID is the specified *resid*. The default is to use a resource ID of 0. (The **-t** option may be used to specify the resource type.)

-p Write SetVersion's version number and the contents of the 'MPST' resource to the diagnostic output file. (You can use the **-d** option just to output the 'MPST' information to the diagnostic output file.)

-prefix *prefix* Set the prefix string on the version. The *prefix* may be any sequence of characters that does not end with a digit (0–9) or a blank.(A blank could be inserted by choosing an appropriate -**fmt** format with leading blanks for the version number.) Once the prefix is set, you can change it only by specifying another **-prefix** string. Alternatively, you can remove the prefix by specifying the *prefix* as a period (.).

-[**p**]**source** *file* Update the string constant in the Pascal source specified by the *file*. The constant is set to be the same as that specified by the 'MPST' resource string in the application or tool. It is assumed that the constant is defined in a **CONST** section somewhere in the first 12800 characters (25 512-byte blocks) of the file as follows:

```
Version = 'ver.rev';ΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔ{some comment}
```

The Δ's indicate required spaces (Spaces or tabs may surround the "="). There may be any number of spaces before the *required* comment. However, because SetVersion edits the line in-place, there must be enough room to allow for changes in the size of the version and revision values—otherwise an error will be reported to the diagnostic file. Case is ignored, and Pascal comments are skipped, when searching for the "Version" identifier in the source. The **-verid** may be used to search for a different identifier if desired.

-r Increment the revision by 1.

-rezsource *file* Update the 'MPST' resource definition in the Resource Compiler source specified by the *file*. The definition is set to be the same as that specified by the 'MPST' resource string in the application or tool. It is assumed that the definition is somewhere in the first 12800 characters (25 512-byte blocks) of the file and is specified as follows:

```
type 'MPST' as 'STRΔ';
resourceΔ'MPST' (0) {
    "Version ver.rev"ΔΔΔΔΔΔΔΔΔΔΔΔΔΔ/*some comment*/
};
```

The Δ's indicate required spaces. There may be any number of spaces before the *required* comment. However, because SetVersion edits the line in-place, there must be enough room to allow for changes in the size of the version and revision values—otherwise an error will be reported to the diagnostic file. Case is ignored, and Rez comments are skipped, when searching for the characters "resourceΔ'MPST'" in the source. Note that, because this is a resource definition and destined to be placed in the application's or tool's resource fork, this option defines the actual string resource that SetVersion will seek in the application or tool. The "Version" in the string here is fixed, and *not* controlled by the
-verid option.

-sr *revision* Set the revision to the specified *revision* integer value.

-suffix *suffix* Set the suffix string on the revision. The *suffix* may be any sequence of characters that does not begin with a digit (0-9). Once the suffix is set, it can be changed only by specifying another -**suffix** string, or removed by specifying the *suffix* as a period (.).

-sv *version* Set the version to the specified *version* integer value.

-t *type* Use the specified resource *type* instead of 'MPST'. (The **-i** option can be used to specify the resource ID.)

-v Increment the version by 1.

-verid *identifier*

Use the specified constant identifer when searching for the **-[p]source** CONST identifier or **-csource** #define identifier.

Example

```
setversion -d -sv 1 -r Asm -psource GlobalDcls -rezsource Asm.r
```

Increments the revision for the MPW Assembler (**-r**) in the resource fork of the file Asm. The version is fixed at 1 (**-sv**), so that Asm will display the version and revision as "1.*rev*" The Pascal `include` file, GlobalDcls, contains the Assembler's global declarations, including the Version string. This include file is updated to match the 'MPST' resource (**-psource**). The resource definitions for the Assembler, in Asm.r, will be similarily updated (**-rezsource**). Finally, this command displays the new version of the diagnostic output file (**-d**).

Choose --- choose or list network volumes and printers

Syntax Choose *[options...] name...*

Description Choose non-interactively mounts or lists the specified AppleShare volumes or printers. Each name takes the form:

```
[zone]:[server[:volume]]
```

("Server" means any file or printer server). The zone name is always optional, and defaults to the current zone. A server name must be preceeded by (at least) a colon. Volume names are only applicable to file servers.

When mounting file server volumes, a server name is required. If a volume name is specified then only that volume is mounted. If the volume name is omitted, or if it is the wildcard character "≈", then all volumes on the server are mounted:

```
[zone]:server:volume
[zone]:server[:≈]
```

When **-list** is specified, the wildcard character "≈" may be used in place of names in all of the fields: "≈" in the zone field expands to all zones; "≈" in the server field expands to all servers in the specified zones; "≈" in the volume-name field expands to volumes on the specified servers (listing volumes on a server requires a server login, i.e. as a user with a valid password or as a guest).

-list also expands the next unspecified item in a name. A zone name followed by nothing else will expand to a list of servers in that zone, and a server name followed by nothing else will expand to a list of volumes on the server.

If a "≈", ":" or "∂" character appears in a server, volume or zone name, it may be quoted with the quote character "∂". This quoting mechanism is in addition to quoting already performed by the shell.

Any number of volumes may be mounted (though there is actually a system-dependent limit on the number of active server connections). Only one printer may be chosen at a time, since only one printer can be active.

Server and volume passwords are case-sensitive. More than one server and volume may be mounted with a single command, but the server and volume passwords must be the same for each, since at most one password of each type may be specified on the command line.

Input None.

Output If **-list** is specified, the names of zones, servers and volumes on file servers are printed in a form suitable for re-input to Choose command lines. If **-c** is specified, the name of the tool (plus appropriate options) appears on each output line.

If **-v** is specified, the names of volumes that were mounted are printed.

If **-cp** is specified, the name,type and driver of the currently chosen printer is printed.

Diagnostics Errors are written to diagnostic output.

Various confusing messages (such as "No AFPLogin call has been successfully made for this session") are usually the result of a missing or mis-typed password.

Status The following status codes are returned:

0	no errors
1	syntax error on command line
2	abort (Command-.)
3	any other error

Options

-list Print information about the specified network entities.

-c Preceed each line of **-list** output with the name of the Choose tool (i.e. output Choose commands).

-type *typename*
This option sets the type of the network object to choose or list. The type name is not case sensitive. For mounting or listing volumes, the type name defaults to 'AFPServer'; for choosing or listing printers, it defaults to the name of the current printer driver (e.g. 'LaserWriter'). Use this option to choose or list network entities of other types.

A type name of "≈" or "=" matches all network entity types. It is possible to list or attempt to mount network entities that are not chooseable. For instance, it is not possible to mount or list volumes on servers of types other than 'AFPServer'.

-p Writes Choose's version number and blow-by-blow progress information to standard output. This is reassuring when doing listings which can take several minutes (e.g. every server on the internet).

The following options are applicable to file servers only, and may not be specified in conjunction with any printer options:

-u *name*
Specify the user name for the server log-in. This option has precedence over the shell variable "{User}", which in turn has precedence over the user name string in the system resource file ('STR ' -16096). If no valid user name is found in any of the above locations, then **-guest** is assumed.

-guest Log-in as a "guest" instead of with a user name.

-pw*password*
Specify the server log-in password. The server password defaults to the value of the shell variable "{ServerPassword}".

-vp *password*
Specify the volume log-in password. The volume password defaults to the value of the shell variable "{VolumePassword}".

-v Print the volume names (only) of any volumes mounted. Colons are appended to each volume name. This is useful in shell scripts when volume names are not known ahead of time.

The following options are applicable to printers only, and may not be specified in conjunction with any file server options:

-pr Specify that a printer is being chosen or listed.

-cp Print the name and type of the currently chosen printer on standard output. This occurs before any new printer is chosen.

-dr *drivername*
Specify the driver name of the printer to choose. This is the name of a printer driver in the system folder (e.g. "ImageWriter").

Examples

```
Choose :Linker:Sources
```

Mount the volume "Sources" on the server "Linker", which is located in the current zone, using the default user name, server password and volume password.

```
Choose -v -guest 'Systems:Sources:Doc' 'Systems:Games:≈'
```

Mount the volume "Doc" on the server "Sources" and every volume on the server "Games" in the zone "Systems" as a guest. Print the names of the volumes that are mounted by the command.

```
Choose -list 'Whale Zone:≈' 'Whale Zone:Moby Dick:≈' '≈:'
```

List all file servers in the zone "Whale Zone", all volumes on the file server "Moby Dick" in that zone (after logging-in with the default user name and server password). List the names of all zones.

```
Choose -pr -list ':≈'
Choose -cp -pr "Zarf:Kitchen Sink"
```

List all printers of the current type in the current zone. Print the name of the currently selected printer, then select the printer called "Kitchen Sink" in the zone "Zarf".

```
Choose -list -type "Fortune Cookie Server" '≈:≈'
```

List all network entities of type "Fortune Cookie Server" in all zones.

See also Unmount and Volumes commands.

Changes to Choose (version 1.0 D3) 15-Jan-1987

1. Most of the switches have been renamed (for brevity and consistency with other tools).

2. The "{UserName}" variable has been renamed to "{User}" for consistency with Projector.

3. The naming syntax for network entities has been completely changed. The new syntax ("zone:volume:server" instead of "server:volume@zone") also permits "wildcard" expansion for listing and mounting, eliminating the need for shell loops to list entities in different zones or servers.

   If you have any shell scripts that used the original version of Choose (1.0 D1) you will have edit them a little before they'll work again.

4. Printer support is included (**-pr**, **-cp** and **-dr** options).

Known Bugs and Features

1. Choose will work only with AppleShare 1.1; when the network group releases AppleShare 2.0 some minor changes will be made.

2. You cannot choose non-AppleTalk printers (e.g. ImageWriters) yet.

DumpFile - display contents of an arbitrary file

Syntax DumpFile [*option*...] *filename*

Description DumpFile permits the user to display the contents of the resource fork or the data fork of a file in a variety of formats.

Input DumpObj does not read standard input.

Output DumpObj writes formatted object file records and disassembled code to standard output.

Diagnostics Errors and warnings are written to diagnostic output. Progress information is also written to diagnostic output with the **-p** option.

Status DumpObj may return the following status values:

0	No problem
2	Fatal error
3	User interrupt

Options

-rf	display the resource fork of the file. (Default is data fork.)
-a	Suppress display of ASCII character values.
-h	Suppress display of hexadecimal characters.
-o	Suppress display of file offsets.
-w nn	Width - display nn bytes on each line of output.
-g nn	Group nn bytes together without intervening spaces.
-p	Write progress information (such as the name of the file being dumped and the version of DumpFile) to diagnostic output.

-r *byte1*[,*byteN*] Display only the byte range from byte1 to byteN.

Examples

```
DumpFile -p ATestFile
```

Formats the data fork of the file *ATestFile* and writes its contents to standard output. This output has the following format:

```
DumpFile -p ATestFile
MPW File Display Utility Version 3.0A1 Release April 15, 1988   Start: 1:24:09

 Copyright Apple Computer, Inc. 1985-1988
 All Rights Reserved.

File : ATestFile
Data Fork Length     : 20
Resource Fork Length : 382
Dumping Data Fork from offset 0 to 20

   0: 54 68 69 73 20 69 73 20 61 20 74 65 73 74 20 66 This.is.a.test.f
  10: 69 6C 65 2E                                     ile.
DumpFile completed normally

Execution required 0 seconds.
```

`DumpFile -w 12 -g 4 ATestFile`

Formats the data fork of the file *ATestFile* and writes its contents to standard output, grouping four bytes at a time and displaying 12 bytes per line. This output has the following format:

```
File : ATestFile
Data Fork Length     : 20
Resource Fork Length : 382
Dumping Data Fork from offset 0 to 20

   0: 54686973 20697320 61207465 This.is.a.te
   C: 73742066 696C652E          st.file.
```

`DumpFile -rf -r 0,30 -g 4 ATestFile`

Formats the resource fork of the file *ATestFile* and writes the contents of bytes 0 through 30 to standard output in four byte groups. This output has the following format:

```
File : ATestFile
Data Fork Length     : 20
Resource Fork Length : 382
Dumping Resource Fork from offset 0 to 30

   0: 00000100 0000014C 0000004C 00000032 .......L...L...2
  10: 696C652E 6F727920 2227227B 646972   ile.ory."'"{dir
```

WhereIs—search for files in directory tree

Change History

Changed -m option to -c for "completely match".
Made -d option include rather than exclude directories. The default is now_files only.

Syntax WhereIs [-c] [-d] [-v] [-s dir] *pattern*

Description Find the location of all files that contain *pattern* as part of their file name. WhereIs can be used to find files hidden in the directory tree. *Pattern* is a full or partial file name. For example, a pattern of "test" will match TestProg.c, test.c, and Work:OutputTest. WhereIs will start searching in the root directory of the default volume and search the entire disk. To constrain the search to a portion of a disk, or to specify different disks, or multiple disks, use the **-s** option. To list any directories contain *pattern*, use the **-d** option. To constrain the search to files that completely match *pattern*, use the **-c** option. The **-v** option will print the number of items matched with *pattern*. Matching is not case sensitive, and regular expressions are not supported.

WhereIs will list the full path name of all files and directories found. Files that contain special characters will be quoted.

Type File Management

Input None.

Output The full path name of any file that contains *pattern* is written to standard output. Also, the total number of files and directories found is written to standard output.

Diagnostics Errors are written to diagnostic output.

Status WhereIs may return the following status values:

0	No errors
1	Syntax error
2	File system error during processing
3	No matches were found

Options

-c	List only files that match *pattern* completely. (In other words, treat *pattern* as a filename.)
-d	Match directories also.
-v	Print a summary line that counts the number of items matched.

`-s dir` Normally, WhereIs starts searching in the root directory of the default volume. This option constrains the search to start in *dir*. Multiple starting directories may be specified. (Each directory must be preceded by **-s**.) Since searching a large hard disk may take several minutes, this option can be used to speed up the search when you know the general location of a file.

Examples

```
WhereIs test
```

Find all files that have "test" in their file name. The output would be something like:

```
   HD:MPW:test.c
   HD:MPW:test.c.o
   HD:MPW:TestMenu.c
   HD:MPW:TestProg.p
```

```
WhereIs -c test.c
```

Find files named test.c. The output (with the same files as the example above) would be:

```
   HD:MPW:test.c
```

```
WhereIs -d test
```

Find all files or directories that have "test" in their leaf name. The output would be:

```
   HD:MPW:TestDir:
   HD:MPW:test.c
   HD:MPW:test.c.o
   HD:MPW:TestMenu.c
   HD:MPW:TestProg.p
```

```
WhereIs -s HD:MPW -s Disk2:Work test
```

Find all files that have "test" in their path name. Search for the files starting in HD:MPW and also in Disk2:Work.

Sort -sort or merge lines of text

Syntax Sort [*options*...] [*files*...]

Description Sort sorts or merges the specified files and prints the result on the standard output. If no input files are specified, standard input is assumed.

Fields and Field Specifications

The **-f** option (see "Options" below) is followed by a comma-seperated list of field specifications. Lines are sorted by extracting and comparing the fields in the order specified until a comparison yeilds inequality. If a field exists in one line, but not the other, the line that possesses the field "wins". If neither line has a field, the lines are considered equal. Fields not sorted on will be output in "random" order (Sort is not a stable sort).

Each of the field specifications takes one of the forms:

```
[F] [.C] [-K] [modifiers]
[F] [.C] [+N] [modifiers]
```

'F' is a field number, '.C' and '-K' are column numbers, and '+N' is a character count. Any of the items may be omitted, as long as at least one item appears. '-K' and '+N' are mutually exclusive. Spaces may appear anywhere in the specification (except within numbers), but they will require shell quoting.

Fields are numbered from 1. A field is a string of characters surrounded by newlines or field seperator characters (usually whitespace; see the **-fs** option). Typically field 1 would be the first word on the line, field 2 the second word, and so on. Field 0 represents the entire line, and is the default if a field number is not specified. Field seperator characters are treated as normal text (not seperators) in field 0.

Columns are numbered from 1. If '.C' is specified, it represents a starting offset into the field, taking into account the (file-dependent) varying width of tab characters, if necessary. '.C' defaults to 1 if it is not specified.

If '-K' is specified it represents the last column to be included in the field. It defaults to "infinity" (the maximum K possible) if not specified. Except for field 0, fields are always terminated by field-seperator characters, so a large K does not mean "the rest of the line".

If '+N' is specified, it represents the number of characters to be included in the field (this differs from '-K' in that tabs are always counted as single characters). It defaults to "infinity" (the maximum N possible) if not specified.

Here is a short description of all possible field specifications:

F	The entirety of field F.
F.C	Columns C...∞ in field F.
F.C-K	Columns C...K in field F.
F.C+N	N characters starting at column C in field F.
F-K	Columns 1...K in field F.
F+N	The first N characters in field F.
.C	Columns C...∞ in the whole line.
.C-K	Columns C...K in the whole line.
.C+N	N characters starting at column C in the whole line.
-K	Columns 1...K in the whole line.
+N	The first N characters of the whole line.

A field specification may be followed by one or more modifier characters:

r	Reverse order of comparison (reverses **-r**).
b	Ignore leading blanks (reverses **-b**).
q	Interpret quotes when extracting field (reverses **-quote**).
d x t l u	Treat field as decimal (**d**), hexadecimal (**x**), normal text (**t**), lowercase text (**l**) or uppercase text (**u**). These modifiers are mutually exclusive.

These modifiers override the corresponding command line options on a field-by-field basis (**r**, **q** and **b** flip the meaning of **-r**, **-quote** and **-b**).

When sorting multiple files, each file may have its own tab setting. When comparing column-aligned fields, Sort correctly handles varying-width tabs, even when comparing records from different files.

Input The specified files, or standard input if no files are specified.

Output If sorting or merging, the concatentation of all specified input files, sorted by the specified fields.

If **-check** is specified, no output is generated; the exit code of the tool indicates if the input was pre-sorted or not.

Diagnostics Errors are written to diagnostic output.

Status The following status codes are returned:

0	no errors
1	syntax error on command line
2	abort (Command-period)
3	any other error
4	out of memory
5	input is not sorted

Options

-b Skip leading blanks in each field.

-check Do not sort, but check if the input is already sorted. Exit with status 0 if the input is sorted, exit with status 5 if the input is not sorted. No output is generated

-d Sort fields as decimal numbers. The numbers can be of arbitrary length.

-f *field1*[*field2*...]
Specify fields to sort by. The default field specification is to sort entire lines as text. (See the discussion on field specifications above).

-fs *string*
Specify the field seperator characters. The default field seperators are space, tab, backspace and form-feed. Fields may not cross newlines. This switch completely replaces the default set of seperators with the specified set.

-l Convert characters to lowercase before comparing them.

-merge Assume each input file is already sorted, and merge the input files into the output file. If one or more of the input files is not sorted, the output will not be sorted.

-o *file* Specify the output file (default is standard output). With this option it is possible to sort (though not to merge) a file "in place"; the output file may be one of the input files.

-p Print version and progress information.

-quote Field extraction is modified by ignoring field seperators enclosed in single and double quotation marks. Characters preceeded by the shell quote character (∂) are properly escaped. Quotation marks themselves are ignored in comparisons, and sets of alternating quotes (e.g. ' " ' ...stuff... ' " ') may be nested to any depth. If a quote "dangles" (there is no matching quote before the end of the line) then the field extends to the end of the line.

-stdin This option serves as a place-holder for the standard input, making it possible to sort or merge standard input with other files.

-r Reverse order of comparison.

-t Sort fields as text (default).

-u Convert characters to uppercase before comparing them.

-unique
Output lines that are identical (with respect to the fields specified with the **-f** option) are printed only once.

-x Sort fields as hexadecimal numbers (upper or lower case). The numbers can be of arbitrary length. A leading dollar sign ($) or '0x' is ignored as whitespace.

Examples

```
Sort able -stdin baker -o output
```

Sort the files "able", "baker" and the standard input, with output to file "output".

```
Sort -x -f '2.2+8, 1tr' frog
```

Sort the file "frog". The first key to sort on consists of eight characters starting at the second column of the second field, treated as a hexadecimal number. The second key to sort on is merely the text of the first field, in reverse order.

```
Sort -p -merge -u one two three infinity
```

Merge the specified files, treating lowercase characters as uppercase. Print version and progress information.

Macintosh • Programmer's • Workshop
3.0

Rez & DeRez

ERS

Tom Taylor
October 8, 1987

Introduction

With the development of the Macintosh II and color QuickDraw came new resources that have a structure that cannot be fully expressed in the Rez language. The goal of this ERS is to present extensions to the Rez language that would enable these new resources to be fully supported in MPW 3.0.

At this point, Rez is the only scriptable resource maniputation tool supported by MPW. Some resource operations, needed by those "internationalizing" system disks and applications, requiring deleting resources and changing resource attributes. Presently, the only way to do this is with ResEdit, or by some other means. This ERS presents simple extensions to Rez to support deleting resources and changing resource attributes.

Supporting the New Resources

In order to support the new color QuickDraw resources, the concept of labels must be incorporated in the Rez language. Labels allow offsets to be calculated and allow accessing of data at a label.

Labels

Labels have a syntax definition like this:

```
label ::=        character {alphanum}* ':'
character ::=    '_' | A | B | C ...
number ::=       0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
alphanum ::=     character | number
```

Labeled statements are valid only within a resource type declaration. A single label may appear on any statement.

Labels may be used in expressions. When labels are used in expressions, only the identifier portion of the label (everything up to, but excluding, the colon) is used (see the section **Labels in Arrays for more information**). The value of a label is always the offset, in bits, between the beginning of the resource and the position where the label occurs when mapped to the resource data. For example:

```
type 'cool' {
    cstring;
endOfString:
    integer = endOfString;
};

resource 'cool' (8) {
  "Neato"
}
```

In this example, the integer following the cstring would contain: (len("Neato") [5] + null byte [1]) * 8 [bits per byte] = 48.

In some cases, it is desireable to access the actual resource data that a label points to. Several new built-in functions allow access to that data:

`$$BitField(`*`label, startingPosition, numberOfBits`*`)`
Returns the *numberOfBits* (maximum of 32) bitstring found *startingPosition* bits from *label.*

`$$Byte(`*`label`*`)`
Returns the byte found at *label.*

`$$Word(`*`label`*`)`
Returns the word found at *label.*

`$$Long(`*`label`*`)`
Returns the longword found at *label.*

For example, the resource type 'STR ' could be redefined without using a pstring:

```
type 'STR ' {
len:  byte = (stop - len) / 8 - 1;
      string[$$Byte(len)];
stop: ;
};
```

Labels in Arrays

Labels declared inside arrays can have many values. For every element in the array, there is a corresponding value for each label defined inside the array. To access the individual values of these labels, array subscripts are used. The subscript values range from 1 to *n* where *n* is the number of elements in the array. Labels within arrays nested in other arrays require multidimensional subscripts. Each level of nesting adds another subscript. The rightmost subscript various most quickly. For example:

```
type 'test' {
       integer = $$CountOf(array1);
      array array1 {
             integer = $$CountOf(array2);
            array array2 {
foo:            integer;
            };
      };
};

resource 'test' (128) {
      {
            {1,2,3},
            {4,5}
      }
};
```

In the example above, the label foo would take on these values:

foo[1,1] = 32	$$Word(foo[1,1]) = 1
foo[1,2] = 48	$$Word(foo[1,2]) = 2
foo[1,3] = 64	$$Word(foo[1,3]) = 3
foo[2,1] = 96	$$Word(foo[2,1]) = 4
foo[2,2] = 112	$$Word(foo[2,2]) = 5

A new built-in function may be helpful in using labels within arrays:

`$$ArrayIndex(`***`arrayname`***`)`

Returns the current array index of the array *arrayname*. An error will occur if this function is used anywhere outside the scope of the array *arrayname*.

Label Limitations

Understanding the fact that Rez and DeRez are basically one-pass compilers will help you understand some of the limitations of labels:

• In order to derez a given type, that type must not contain any expressions that contain more than one undefined label. An undefined label is a label that occurs lexically after the expression. Using a label in an expression before the label is defined, will define the label. The example belows demonstrates how expression can only have only one undefined variable:

```
type 'test' {
   /* In the expression below, start is defined, next is undefined. */
start:   integer = next - start;

   /* In the expression below, next is defined because it was used
        in a previous expression, but final is undefined. */
middle:  integer = final - next;
next:    integer;
final:
};
```

Actually, Rez can compile types that have expressions containing more than one undefined variable, but DeRez will not be able to decompile those resources and will simply generate data resource statements.

• The label specified in $$BitField(), $$Byte(), $$Word(), and $$Long() expression must occur lexically before the expression, otherwise an error is generated.

Examples

Example #1

This is the modified 'ppat' declaration using the proposed Rez labels (everything that is changed is bold). Previously, the whole end section of the resource had to be combined into a single hex string (everything below the PixelData label). Using labels, the complete 'ppat' definition can now be expressed in Rez.

```
type 'ppat' {
        /* PixPat record */
        integer    oldPattern,                        /* Pattern type          */
                   newPattern,
                   ditherPattern;
        unsigned longint = PixMap / 8;                /* Offset to pixmap      */
        unsigned longint = PixelData / 8;             /* Offset to data        */
        fill long;                                    /* Expanded pixel image  */
        fill word;                                    /* Pattern valid flag    */
        fill long;                                    /* expanded pattern      */
        hex string [8];                               /* old-style pattern     */

        /* PixMap record */
PixMap:
        fill long;                                    /* Base address          */
        unsigned bitstring[1] = 1;                    /* New pixMap flag       */
        unsigned bitstring[2] = 0;                    /* Must be 0             */
        unsigned bitstring[13];                       /* Offset to next row    */
        rect;                                         /* Bitmap bounds         */
        integer;                                      /* pixMap vers number    */
        integer    unpacked;                          /* Packing format        */
        unsigned longint;                             /* size of pixel data    */
        unsigned hex longint;                         /* h. resolution (ppi) (fixed) */
        unsigned hex longint;                         /* v. resolution (ppi) (fixed) */
        integer       chunky, chunkyPlanar, planar;   /* Pixel storage format  */
        integer;                                      /* # bits in pixel       */
        integer;                                      /* # components in pixel */
        integer;                                      /* # bits per field      */
        unsigned longint;                             /* Offset to next plane  */
        unsigned longint = ColorTable / 8;            /* Offset to color table */
        fill long;                                    /* Reserved              */

PixelData:
        hex string [(ColorTable - PixelData) / 8];

ColorTable:
        unsigned hex longint;                         /* ctSeed                */
        integer;                                      /* transIndex            */
        integer = $$Countof(ColorSpec) - 1;     /* ctSize           */
        wide array ColorSpec {
            integer;                            /* value            */
            unsigned integer;                   /* RGB: red         */
            unsigned integer;                   /*      green       */
            unsigned integer;                   /*      blue        */
        };
};
```

Example #2

This is another example of a new resource definition with the new features in bold. This example uses the $$BitField() function to access information store in the resource to calculate the size of the various data areas tacked onto the end of the resource. Previously, all of data had to be combined into one hex string.

```
type 'cicn' {
        /* IconPMap (pixMap) record */
        fill long;                                    /* Base address          */
        unsigned bitstring[1] = 1;                    /* New pixMap flag       */
        unsigned bitstring[2] = 0;                    /* Must be 0             */
pMapRowBytes: unsigned bitstring[13];                 /* Offset to next row    */
Bounds: rect;                                         /* Bitmap bounds         */
        integer;                                      /* pixMap vers number    */
        integer unpacked;                             /* Packing format        */
        unsigned longint;                             /* Size of pixel data    */
        unsigned hex longint;                         /* h. resolution (ppi) (fixed) */
        unsigned hex longint;                         /* v. resolution (ppi) (fixed) */
        integer     chunky, chunkyPlanar, planar;     /* Pixel storage format  */
        integer;                                      /* # bits in pixel       */
        integer;                                      /* # components in pixel */
        integer;                                      /* # bits per field      */
        unsigned longint;                             /* Offset to next plane  */
        unsigned longint;                             /* Offset to color table */
        fill long;                                    /* Reserved              */

        /* IconMask (bitMap) record */
        fill long;                                    /* Base address          */
maskRowBytes: integer;                                /* Row bytes             */
        rect;                                         /* Bitmap bounds         */
        /* IconBMap (bitMap) record */
        fill long;                                      /* Base address         */
iconBMapRowBytes: integer;                              /* Row bytes            */
        rect;                                           /* Bitmap bounds        */

        fill long;                                      /* Handle placeholder   */

        /* Mask data */
        hex string [$$Word(maskRowBytes) * ($$BitField(Bounds, 32, 16) /*bottom*/
               - $$BitField(Bounds, 0, 16) /*top*/)];

        /* BitMap data */
        hex string [$$Word(iconBMapRowBytes) * ($$BitField(Bounds, 32, 16) /*bottom*/
               - $$BitField(Bounds, 0, 16) /* top */)];

        /* Color Table */
        unsigned hex longint;                           /* ctSeed               */
        integer;                                        /* transIndex           */
        integer = $$Countof(ColorSpec) - 1;             /* ctSize               */
        wide array ColorSpec {
               integer;                                 /* value                */
               unsigned integer;                        /* RGB: red             */
               unsigned integer;                        /*      green           */
               unsigned integer;                        /*      blue            */
        };

        /* PixelMap data */
        hex string [$$BitField(pMapRowBytes, 0, 13) *
               ($$BitField(Bounds, 32, 16) /* bottom */
               - $$BitField(Bounds, 0, 16) /*top*/)];
};
```

Making Rez a More General Resource Manipulator

Right now, Rez is just a resource compiler and a resource mover. By adding two simple functions, Rez can be a more general purpose resource manipulator.

The first new function is the ability to delete resources from an existing resource file.

> **NOTE:** It must be stated right off that these two new functions are only valid when the -append option is specified. It makes no sense to delete resources while creating a new resource file from scratch.

The syntax of the delete statement is this:

```
delete resource-type ['('resource-name| ID[:ID]')'];
```

> Delete the resource of type *resource-type* from the output file with the specified resource name or resource ID range. If the resource name or ID is omitted, all resources of type *resource-type* are deleted.

The second new function is the ability to change a resource's attributes. The syntax of the change statement is this:

```
change resource-type1 ['('resource-name| ID[:ID]')'] to
   resource-type2 '('ID[, resource-name] [,attributes…]')';
```

> Change the resource of type *resource-type1* from the output file with the specified resource name or resource ID range to a resource of type *resource-type2* with the specified ID. You can optionally specify a resource name and resource attributes. If the resource name or attributes are not specified, the name and attributes are not changed.

Macintosh® ResEdit Reference

Beta Version
18 July, 1988

Apple® Technical Publications

This document contains preliminary information. It does not include

- final technical information
- a glossary

2/21/88

Contents

Chapter 1
ResEdit Overview

This chapter introduces ResEdit™, a stand-alone application for editing resources.

ResEdit is an interactive, graphically based application for manipulating the various resources in a Macintosh® file. (Some Macintosh files don't have any resources, but all applications and most of the System Folder files do.) If you are used to other computers, you will rapidly discover that resources are handled very differently on the Macintosh computer. They are kept distinct from data in a special part of the file (the resource fork) and are separated from each other in small packages so that individual resources can be examined and edited easily.

There are many different types of resources, and you can create your own with ResEdit if you don't find the type you need. It lets you create and edit all standard resource types except 'NFNT' and 'snd ', and copy and paste all resource types (including 'NFNT' and 'snd '). ResEdit actually includes a number of different resource editors: There is a general resource editor for editing any resource in hex and ASCII format, and there are several individual resource editors for specific resource types. Many resource templates are predefined. You can also create your own resource editors or templates to use with ResEdit. Thus, for example, you could build a template that would permit you to do some editing of 'snd ' resources. It is not particularly useful to build a template for 'snd ' resources; nor is attempting to edit 'CODE' resources with the general resource editor — there are better ways to accomplish these ends!

Uses

ResEdit is especially useful for creating and changing graphic resources such as dialog boxes and icons. For example, you can use ResEdit to put together a quick prototype of a user interface and try out different formats and presentations of resources. Anyone can quickly learn to use ResEdit for translating resources into languages other than English without having to recompile programs. You can use ResEdit to modify a program's resources at any stage in the process of program development.

Extensibility

A key feature of ResEdit is its extensibility. Because it can't anticipate the formats of all the different types of resources that you may use, ResEdit is designed so that you can teach it to recognize and parse new resource types.

There are two ways that you can extend ResEdit to handle new types:

- ☐ You can create templates for your own resource types. ResEdit lets you edit most resource types by filling in the fields of a dialog box—this is the way you edit 'BNDL' and 'FREF' resources, for example. The layout of these dialog boxes is determined by a template in ResEdit's resource file, and you can add templates to edit new resource types. Resource templates are described later in this manual.
- ☐ You can program your own special-purpose resource picker or editor (or both), and then add it to ResEdit. The *resource picker* is the code that displays all the resources of one type in the

resource type window. The *editor* is the code that displays and allows you to edit a particular resource. These pieces of code are separate from the main code of ResEdit. A set of Pascal or C routines, called ResEd, is available for this purpose—see Chapter 5 for information.

The resource development cycle

ResEdit is often used with Macintosh Programmer's Workshop (MPW) and other program development systems. Once you have created or modified a resource with ResEdit, you can use MPW's Resource Decompiler, DeRez, to convert the resource to a textual representation that can be processed by the Resource Compiler, Rez. You can then add comments to this text file or otherwise modify it with the MPW Shell editor or another text editor. (Rez and DeRez are fully described in the *MPW Reference*.) Figure 1-1 shows how ResEdit fits into the development process with MPW.

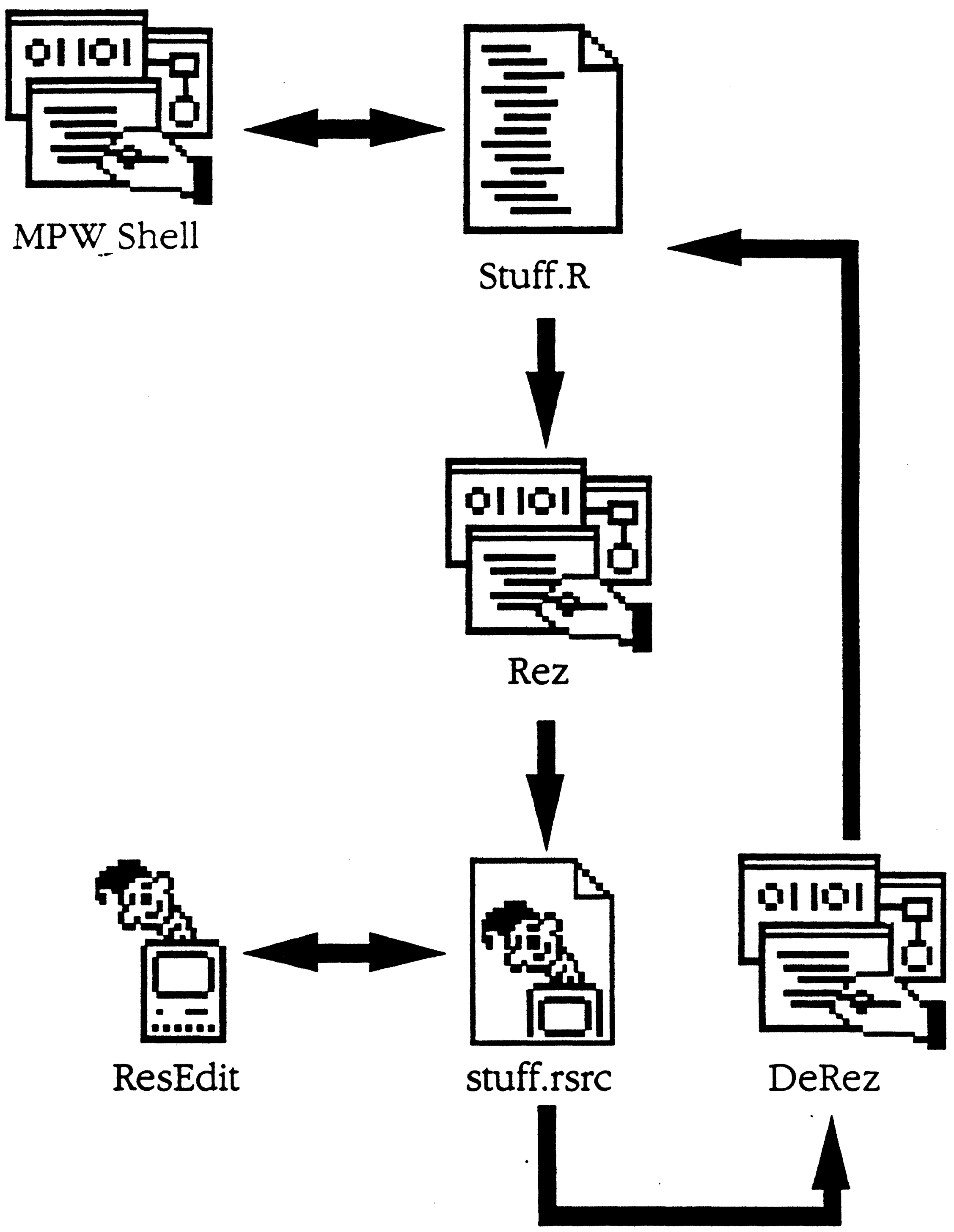


Figure 1-1
The resource development cycle

Overview of resource types

Consider, within the context of ResEdit, that there are three kinds of resources on the Macintosh. Some resources are accessed with individual pickers and edited with individual editors that are part of ResEdit. These resources are described in some detail in Chapter 3.

Within the first category are several pictorial resource types ('ICON', 'ICN#', 'PAT ', and so on). All of the editors for these pictorial resources behave very much like FatBits, but only one (the 'FONT' Editor) offers several tools; for the rest, the cursor acts like the pencil tool in MacPaint. In the 'ICON', 'ICN#', and 'CURS' editors, holding down the Shift key allows you to use the Marquee tool, but only to move part of the picture. In this (1.2b1) release you cannot yet cut, copy, or paste with the Marquee tool.

Other resources are edited as templates. Some information on templates is included in this version of the *ResEdit Reference*, but there wasn't time to include a full description.

Still other resources are edited with thegeneral editor (the HEXA Editor), unless you write your own templates or editors for them. Again, there is not much discussion of this third category in this version of the *ResEdit Reference.*

Resource ID numbers

Within a given resource type, resource ID numbers must be unique. Resources can, in general, have any ID number between −32768 and +32767, but you should be aware of the following restrictions, which apply to most resources:

- ID numbers from −32768 to −16385 are reserved. Do not use them!
- ID numbers from −16384 to −1 are used for system resources that are owned by other system resources.
- ID numbers from 0 to 127 are used for system resources.
- ID numbers from 128 to 32767 are available to you for your uses.

Some system resources own others. The "owner" contains code that reads the "owned" resource into memory. For example, desk accessories ('DRVR' resources) can have their own patterns, strings, and so on. Please see Chapter 5 of *Inside Macintosh*, Volume I, for more information.

For 'FONT' resources, the ID number for a particular resource is the family ID number multiplied by 128, to which is then added the point size of the font. Thus, font 268 is New York (family ID 2) 12 point. The family ID is the ID number of the associated 'FOND' resource. Note that for 'FOND' fonts, the family ID cannot exceed 255. This relationship is quite different for 'NFNT' fonts; there, the ID number of the 'NFNT' resource is arbitrary, so there is no particular connection between the 'FOND' ID number (the family ID) and the ID numbers of the corresponding 'NFNT' resources. (This decoupling facilitates resolution of conflicting ID numbers, which is performed automatically by the Font/DA Mover.) Fonts of type 'NFNT' can be moved and deleted, but not modified, with ResEdit. (Version 3.8 or later of the Font /DA Mover also moves and deletes type 'NFNT'.) 'NFNT' fonts can contain and display more than one bit per pixel, and can be assigned absolute colors with a corresponding 'fctb' resource, which is a ColorTable record. (Font ColorTable records are discussed in *Inside Macintosh*, Volume V, in the section on the Color Manager. The Font Manager is discussed in some detail in *Inside Macintosh*, Volumes IV and V.)

In general, it is a good idea to use the same ID for an 'ALRT' or 'DLOG' and its associated 'DITL', though this is not required.

Chapter 2
Getting Started

If you are new to ResEdit, you will want to proceed with some caution, as ResEdit is remarkably powerful and can easily damage or destroy your files. It is a good idea to edit spare copies of files rather than originals, and to avoid editing the contents of the System Folder on the current startup volume. (Under MultiFinder™, you cannot use ResEdit to open the current Finder™ or desktop file.)

If you are using ResEdit in conjunction with MPW, please read the relevant material in the *MPW Reference*.

Installing ResEdit

If you will be using ResEdit within the MPW Shell, copy ResEdit into the Applications folder or elsewhere in the search path defined by the {Commands} variable. If you will not, the location of ResEdit is not important.

Invoking ResEdit

From the Finder, you can select and open the ResEdit icon. From the MPW Shell, you can start ResEdit by entering either of these commands:

```
ResEdit
ResEdit file1 file2 …
```

The latter command causes ResEdit to open the named files automatically.

ResEdit displays a window that lists the files and folders for each disk volume currently mounted (Figure 2-1).

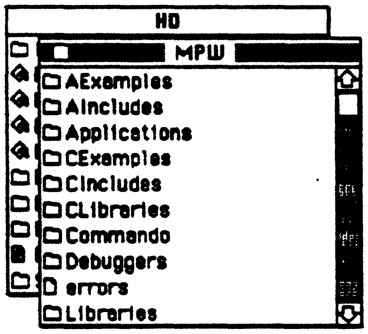


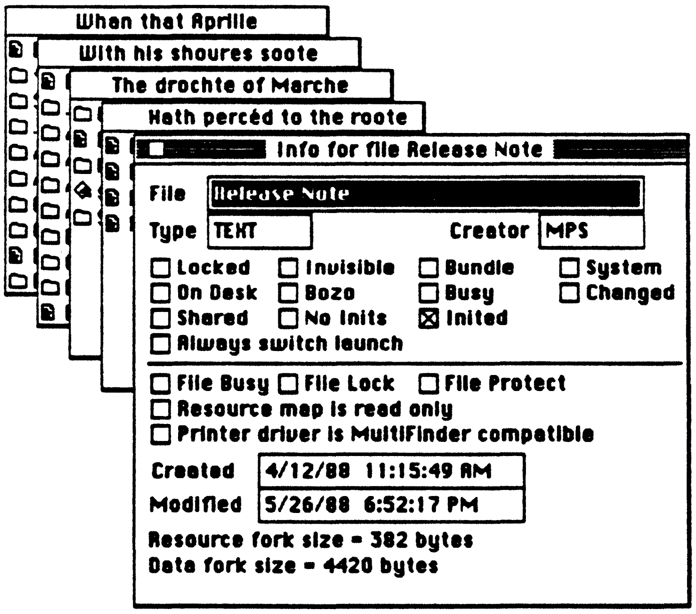
Figure 2-1
Disk volume windows

Working with files

To list the resource types in a file, select and open the filename from the list. (You can select a filename by clicking on it or by typing one or more characters of the filename.) To select more than one item, hold down the Command key while clicking the individual items, or click an item at the beginning of the range you want to select, hold down the Shift key, and click the item at the end of the range. (You can, of course, continue to select or deselect individual items with the Command key.)

When a directory window is the active window, the File menu commands act as follows:

New	Creates a new file.
Open	Opens the selected file or folder. (Choosing this command has the same effect as double-clicking the filename or pressing the Return key or the Enter key.)
Close	Closes the volume window. (Using this command has the same effect as clicking the close box.) If the volume is a 3.5-inch disk, the disk is ejected.
Save	Not usable at this level.
Get Info	Displays file or folder information and allows you to change it. Figure 2-2 is an example of the File Info window.



File: Release Note
Type: TEXT Creator: MPS

☐ Locked ☐ Invisible ☐ Bundle ☐ System
☐ On Desk ☐ Bozo ☐ Busy ☐ Changed
☐ Shared ☐ No Inits ☒ Inited
☐ Always switch launch

☐ File Busy ☐ File Lock ☐ File Protect
☐ Resource map is read only
☐ Printer driver is MultiFinder compatible

Created: 4/12/88 11:15:49 AM
Modified: 5/26/88 6:52:17 PM
Resource fork size = 382 bytes
Data fork size = 4420 bytes

Figure 2-2
A File Info window

Transfer...	Allows you to transfer to an application other than the application that launched ResEdit.
Quit	Quits ResEdit and returns to the Finder (or the MPW Shell, HyperCard, or whatever program launched ResEdit).

Warning
You can edit any file shown in the window, including the System file and ResEdit itself (though there are some restrictions under MultiFinder). It's dangerous, though, to edit a file that's currently in use. Edit a copy of the file instead.

ResEdit recognizes a new disk when it's inserted, and handles multiple disk drives. Note that you can also use ResEdit to delete files:

□ To delete a file, select the file and choose Clear from the Edit menu.

□ To copy a resource file, you must select all of its resources and copy them. Then paste them into a new file. (File attributes are not automatically copied by this operation—you must set them via the Get Info command.) *ResEdit cannot copy a data fork.*

Working within a file

When you open a file, a window displays a list of all the resource types in that file (Figure 2-3). While this window is the active window, you can create new resources, copy or delete existing resources, and paste resources from other files.

❖ *Note:* The resources are displayed by a resource picker. The general resource picker displays the resources by type, name, and ID number; there are also special resource pickers for some resource types. (See Chapter 3 for a detailed discussion of some of these resources.)

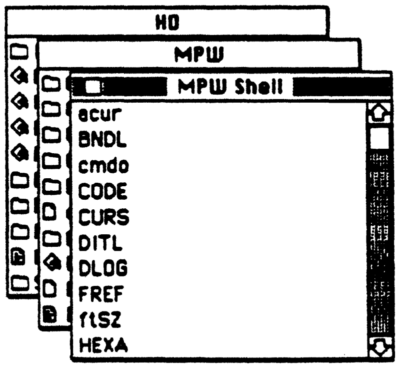


Figure 2-3
A ResEdit file window

When a file window is the active window, the File menu commands have the following effects:

New	Creates a new resource in the open file.
Open	Opens a window displaying all resources of the resource type selected. (Select the resource type by clicking on it or by typing its first character, if that's unique, or first two characters.)

❖ *Note:* If you hold down the Option key while opening a resource type, the resource window will open with the general resource picker.

Open general	Opens the general resource picker.
Close	Closes the file window and asks if you want to save the changes you have made.
Save	Saves the changes you have made.

❖ *Note:* If you've made changes, you should not reboot before closing. Under ordinary circumstances, pressing the programmer's switch is the only way to reboot from inside ResEdit, so most ResEdit users are unlikely to have any problems related to this issue.

Revert	Changes the resource file back to the version that was last saved to disk.

Warning

If you have cut or cleared an individual resource, this level is the *only* level at which choosing Revert will accurately restore the file!

Transfer...	Quits ResEdit, allowing you to transfer control to another application.
Quit	Quits ResEdit. If you have made any changes, ResEdit asks whether you want to save them.

When a file window is the active window, the Edit menu commands have the following effects:

Cut	Removes all resources of the resource types selected, placing them in the ResEdit scrap.
Copy	Copies all resources of the resource types selected into the ResEdit scrap (not the Clipboard).
Paste	Copies the resources from the ResEdit scrap into the file window's resource type list.

Clear	Removes all resources of the resource type selected, without placing them in the ResEdit scrap.
Duplicate	Not usable here.

Working within a resource type

Opening a resource type produces a window that lists each resource of that type in the file (Figure 2-4). This list will take different forms, depending on the particular resource picker; if you hold down the Option key during the open, the general resource picker is invoked.

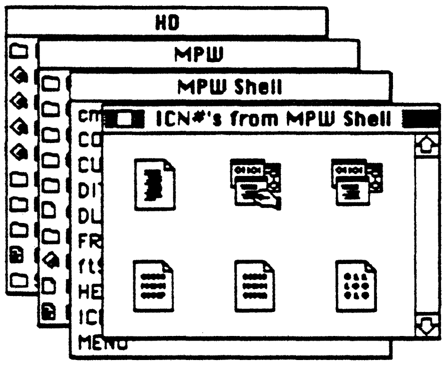
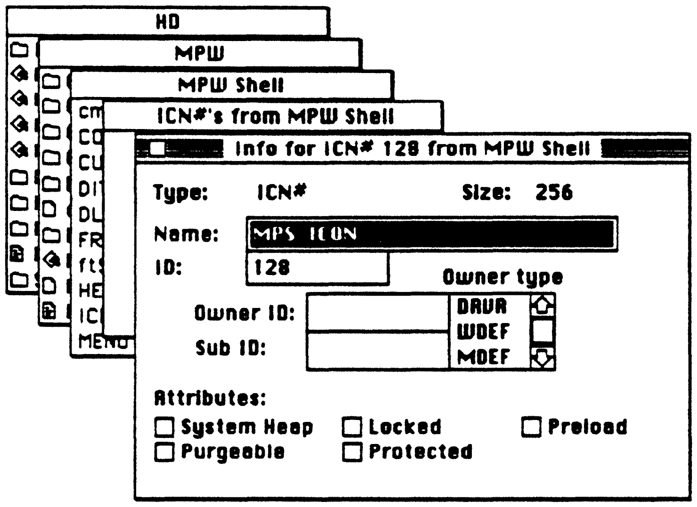


Figure 2-4
A resource type window (with custom picker)

When a resource type window is the active window, the File menu commands have the following effects:

New	Creates a new resource and opens its editor.
Open	Opens the appropriate editor for the resource you selected.
Open As	Lets you open a resource using a template you specify.
Open general	Opens the general (HEXA) resource editor.
Close	Closes the resource type window.
Save	Saves the changes you have made to the file.
Revert	Changes all resources of the open type back to what they were before you opened the resource type window, *unless you have cut or cleared a resource.* (It will restore all but the ones that were cut or cleared.)
Get Info	Displays resource information and allows you to change it. Figure 2-5 is an example.

Transfer...	Quits ResEdit, allowing you to transfer control to another application.
Quit	Quits ResEdit. If you have made any changes, ResEdit asks whether you want to save them.



Type: ICN# Size: 256
Name: MPS ICON
ID: 128
Owner type: DRVR WDEF MDEF
Owner ID:
Sub ID:
Attributes:
☐ System Heap ☐ Locked ☐ Preload
☐ Purgeable ☐ Protected

Figure 2-5
An 'ICN#' Get Info window

When a resource type window is the active window, the Edit menu commands have the following effects:

Undo	Not usable.
Cut	Removes the resources that are selected, placing them in the ResEdit scrap.
Copy	Copies all the resources that are selected into the ResEdit scrap.
Paste	Copies the resources from the ResEdit scrap into the resource type window.
Clear	Removes the resources that are selected, without placing them in the ResEdit scrap.
Duplicate	Creates a duplicate of the selected resources and assigns a unique resource ID number to each new resource.

When you choose Open As, a list of templates is displayed, and you can pick the one you want to use.

❖ *Note:* Using a template with a resource that does not match its definition is improper.

❖ *Note:* If you hold down the Option and Command keys while opening a resource, the effect is the same as choosing Open As.

Chapter 3
Editing Individual Resources

Some of ResEdit's resource editors are discussed in this chapter. The use of the editors not discussed here should be apparent when you run them.

To open an editor for a particular resource, either double-click the resource or select it and choose Open from the File menu. One or more auxiliary menus may appear, depending on the type of resource you're editing. Some editors, such as the 'DITL' editor, allow you to open additional editors for the elements within the resource. All the editors use File and Edit menus similar to those described above, but operate on individual resources or individual elements of a resource, and hence vary in their appearance and function as explained in this chapter.

If you hold down the Option key while opening a resource, the *general data editor* (HEXA Editor) is invoked. This editor allows you to edit the resource as hexadecimal or ASCII data.

❖ *Note:* The general data editor can now edit resources larger than 255K bytes; if a resource is between 256 and 511K, each click in the up or down scroll arrows scroll 2 lines; if between 512 and 767K, each click scrolls 3 lines, and so on. (The scroll bars keep track of position with an integer.)

If you hold down the Option and Command keys while opening a resource, a list of templates is displayed. You may then select the template that is appropriate for the resource you are opening.

'WIND' resources

A 'WIND' resource defines a window on the screen. When you open a 'WIND' resource, ResEdit displays a small picture of the screen with the window shown in its usual size and location, to scale. You can move the window by clicking anywhere except in the lower-right corner, and size it by using its lower-right corner. Moving or sizing a window changes the default values when the window is actually displayed. To change the name of the window, select Display as Text from the WIND menu. (When the window appears on the screen in normal operation, the name may be displayed. If it is displayed, it shows up as a title, in the title bar.)

Firgure 3-1 shows a 'WIND' resource open for editing.

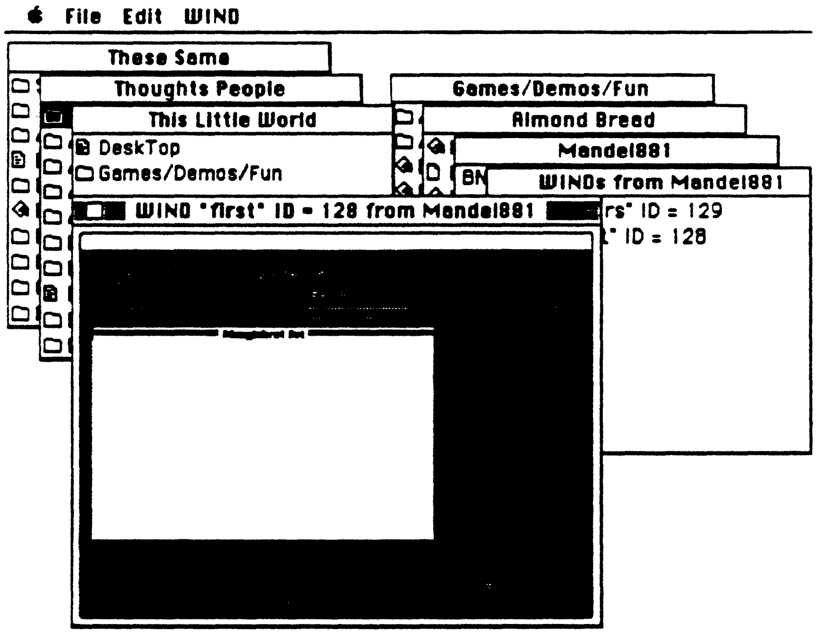


Figure 3-1
Editing a 'WIND' resource

'ALRT' and 'DLOG' resources

'ALRT' and 'DLOG' resources display dialog boxes on the screen. Editing them is much like editing 'WIND' resources, except that if you double-click on the picture of the dialog box after opening the resource, the corresponding 'DITL' resource is automatically opened. (See the next section.) When you display an individual 'ALRT' or 'DLOG' resource, a corresponding menu appears. It has only one item, Display as Text. In the text view, the resource ID of the associated 'DITL' can be changed.

Figure 3-2 shows an 'ALRT' open for editing. You can see the ALRT menu header in the menu bar.

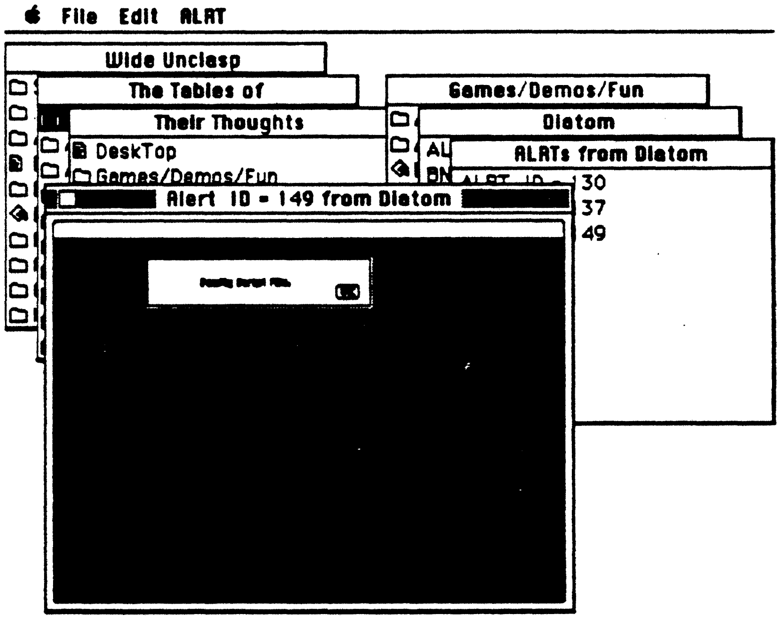


Figure 3-2
Editing an 'ALRT' resource

'DITL' resources

Because they are linked, the 'DITL' (dialog item list) resource for a given item is usually given the same ID number as the parent 'DLOG' or 'ALRT'.

For 'DITL' resources, the editor displays an image of the item list as your program would display it in a dialog or alert box. When you select an item, a size box appears in the lower-right corner of its enclosing rectangle so that you can change the size of the rectangle. You can move an item by dragging it with the mouse.

If you open an item within the dialog box, the editor associated with the item is invoked; for an 'ICON', for example, the icon editor is invoked. If you hold down the Option key while opening a 'CNTL', 'ICON', or 'PICT', the general data editor is invoked. If you hold down the Command key while opening a 'CNTL', 'ICON', or 'PICT', the DITL Item Editor (the same editor used for buttons, static text, and so on) is invoked.

Figure 3-3 shows the 'DITL' corresponding to the 'ALRT' from Figure 3-2. The ALRT menu has been replaced by the DITL menu.

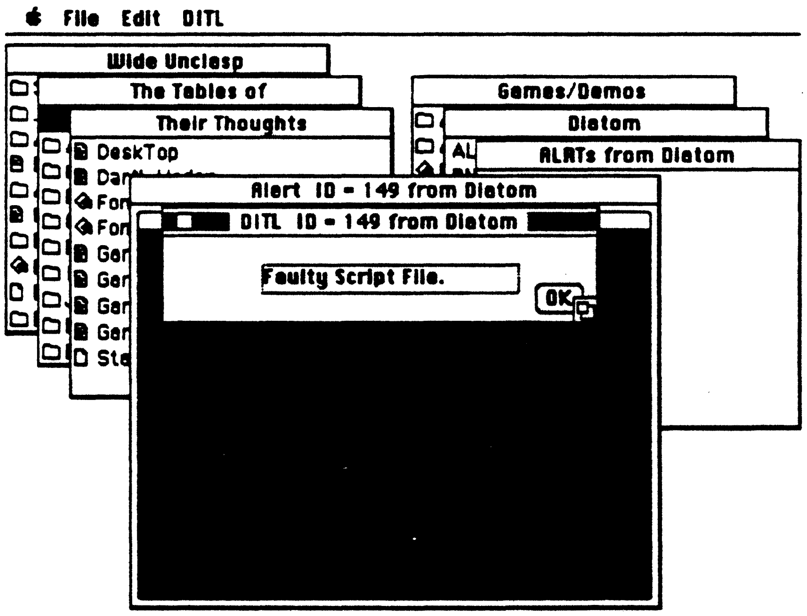


Figure 3-3
Editing a 'DITL' resource

The DITL menu contains the following commands:

Bring to Front	Allows you to change the order of items in the item list. Bring to Front causes the selected item to become the last (highest numbered) item in the list. The actual number of the item is shown by the 'DITL' Item Editor. (Since it is drawn last, it becomes the front item.)
Send to Back	Like Bring to Front, except that it makes the selected item the first item in the list—that is, item number 1. (Since it is drawn first, it becomes the back item.)
Set Item Number	Allows you to specify a new number for the selected item.
Select Item Number	Allows you to select an item by specifying its number.
Align to Grid	Aligns the item on an invisible 8-pixel by 8-pixel grid. If you change the item location while Align to Grid is on, the location will be adjusted such that the upper-left corner lies on the nearest grid point above and to the left of the location you gave it. If you change the size, it will be made a multiple of 8 pixels in both dimensions.
Use RSRC Rectangle	Restores the enclosing rectangle to the rectangle size stored in the underlying resource. Note that this command works on 'ICON', 'PICT', and 'CNTL' items only; the other items have no underlying resources.
Use Full Window	Adjusts the window size so that all items in the item list are visible in the window.
Use Owner Window	Changes the 'DITL' back to the size specified in the parent 'DLOG' or 'ALRT'. The algorithm used to find the parent is as follows:

1. Check for a 'DLOG' with the same ID;
2. Check for an 'ALRT' with the same ID;
3. Check for any 'DLOG';
4. Check for any 'ALRT';
5. Use Full Window.

'CURS' resources

For 'CURS' resources, the editor displays three images of the cursor (Figure 3-4). You can manipulate all three images with the mouse.

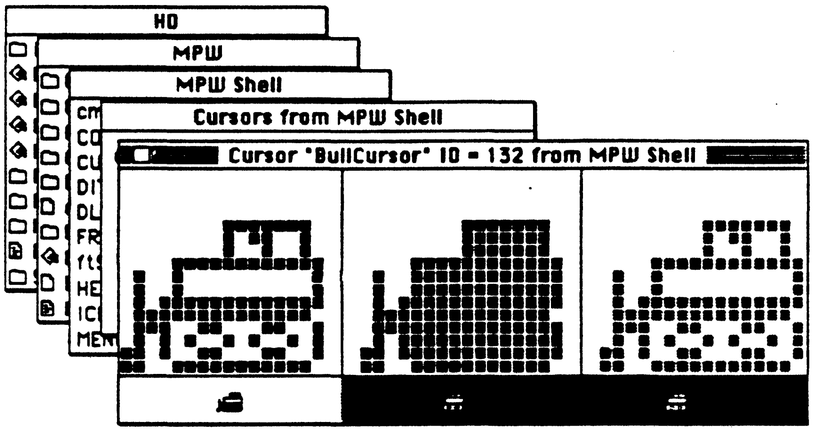


Figure 3-4
Editing a 'CURS' resource

In Figure 3-4, the left image shows how the bulldozer cursor will appear. The middle image is the mask for the cursor, which affects how the cursor appears on various backgrounds. The right image shows a gray picture of the cursor with a single point in black. This point is the cursor's "hot spot" (its active region).

The Cursor menu contains the following commands:

Try Cursor	Lets you try out the cursor by having it become the cursor in use.
Restore Arrow	Restores the standard arrow cursor.
Data -> Mask	Makes a filled-in copy of the cursor in the mask editing area.

'ICON' resources

For 'ICON' resources, the editor displays one panel in the window (Figure 3-5). The left side of this panel shows an enlargement of the icon, and is an editing area. The right side of the panel shows the icon at full scale. Edit as if you were in FatBits.

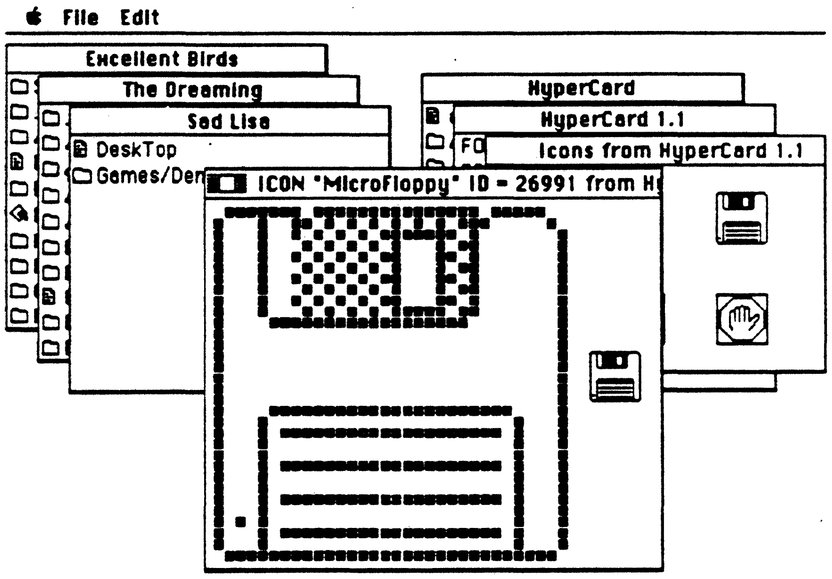


Figure 3-5
Editing an 'ICON' resource

'ICN#' resources

For 'ICN#' resources, the editor displays two panels in the window (Figure 3-6). The upper panel is used to edit the icon. It contains an enlargement of the icon on the left, and an enlargement of the icon's mask on the right. The lower panel shows, from left to right, how the icon will look unselected, selected, and open on both a white and a gray background. It also shows how the icon will appear unselected, selected, and open in the Finder's small icon view.

In recent versions of the Finder, 'ICN#' resources are displayed on the screen as follows: First the mask is used to blank an area of the screen. Then an OR is performed, using the icon as data, in the same screen area. (When a highlighted icon is displayed, the foreground and background colors are swapped before the OR operation is performed on the data.) If the mask is not the same shape as the outline of the icon, the results will in general be unaesthetic unless the background is black.

The Cursor menu contains the following commands:

Data -> Mask	Makes a filled-in copy of the icon in the mask editing area.
Display using old method	Lets you display the icon in the lower panel, using the method that was used by pre-6.0 finders. If the mask is just a filled-in copy of the icon, you probably won't see a difference between the old and new displays.

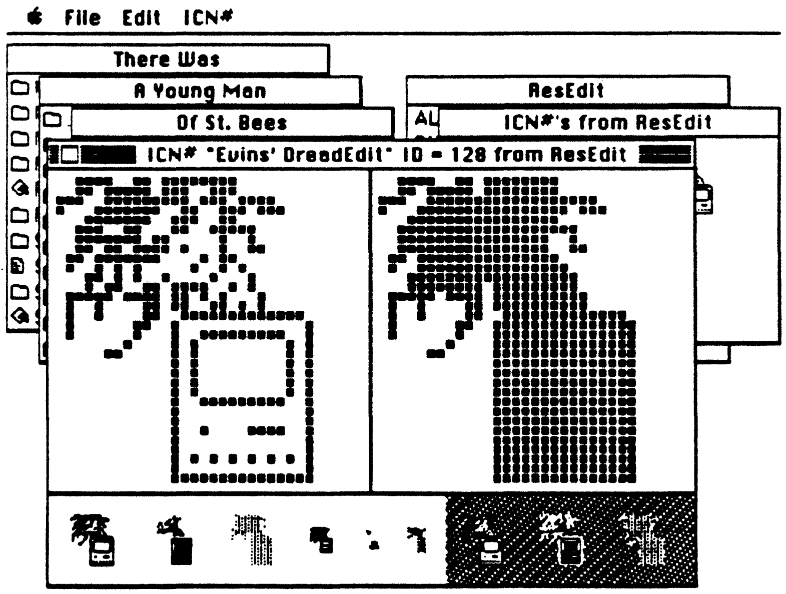


Figure 3-6
editing an 'ICN#' resource

An ICN# editing example

If you have written an application, and you want to install a new icon for it when you already have an old one in the Finder's desktop file, follow these steps:

1. Open the file called DeskTop. You cannot do this under MultiFinder! If you are using MultiFinder, restart your Macintosh while holding down the Command key, to disable MultiFinder temporarily.

2. Open type 'BNDL' and find the bundle that belongs to your application. (This is the one that has your owner name in it.) Look through the bundle and mark down the types ('ICN#', 'FREF') and resource IDs of the resources bundled together by the bundle.

3. Go back to the DeskTop window and remove these resources along with your 'BNDL' and signature resource (the resource whose type is your application's signature).

4. Close the DeskTop window, save changes, and quit ResEdit. Your new icon will be installed if you have the proper 'BNDL', 'FREF', and 'ICN#' resource numberings.

◆ *Note:* To see how 'BNDL', 'FREF', and 'ICN#' resources are interrelated, use ResEdit to look at those resources in an existing application.

Alternatively, you can rebuild the DeskTop file by holding down the Option and Command keys when entering the Finder. (This method is faster and easier, but destroys Finder "Get Info" comments. On a non-HFS volume, it also destroys folder names.)

'SICN' resources

'SICN' (small icon) resources are edited much as other pictorial resources are, but unlike 'ICON' or 'ICN#' resources, they can occur in groups. A typical display is shown in Figure 3-7. The upper panel is enlarged, and shows the icon currently being edited. The lower panel shows three icons at full scale. The one shown in the upper panel is enclosed in a box in the lower panel. To get to a different icon, click on its picture in the lower panel. If the one you want to edit is not currently visible, click on either the righthand or lefthand picture, as appropriate, until it appears.

A new icon can be added before (to the left of) the currently selected icon by selecting the New command from the File menu. Commands on the Edit menu can be used to cut copy, caste, clear, or duplicate icons.

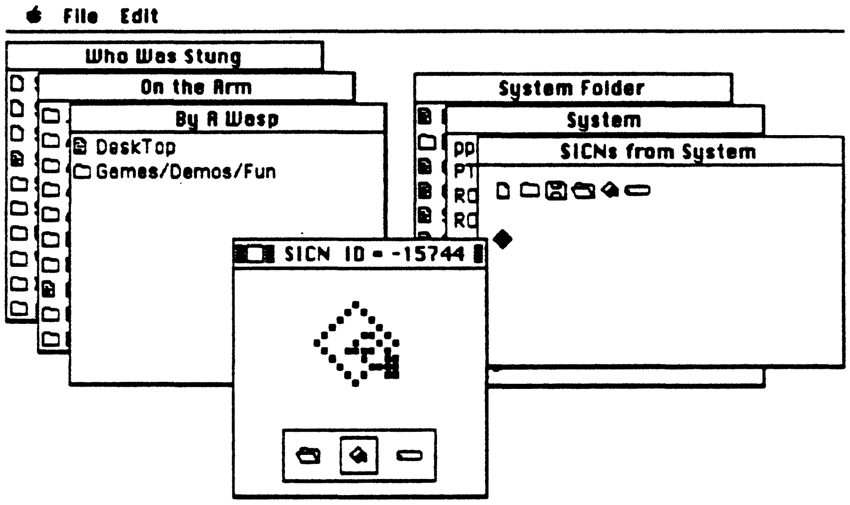
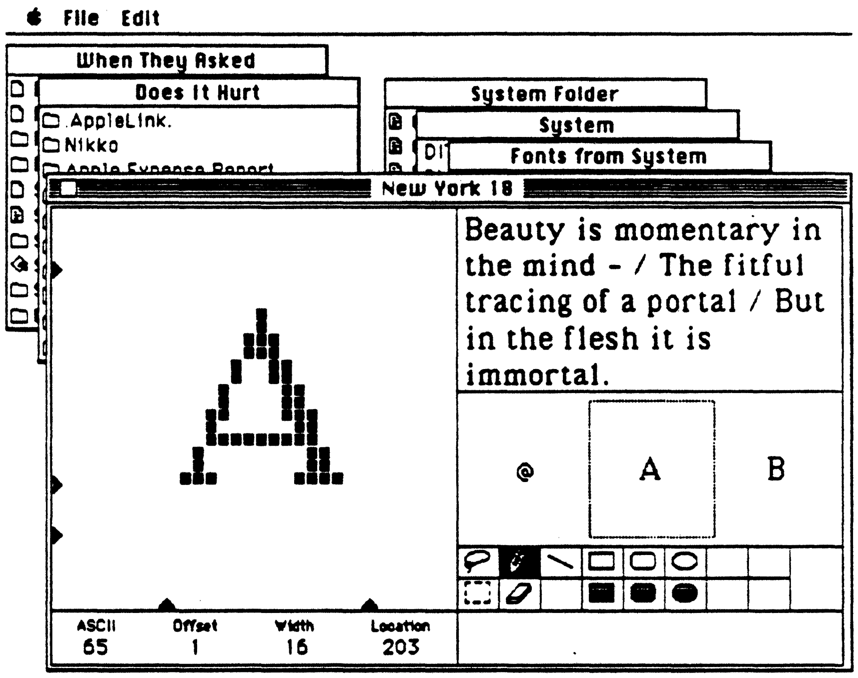


Figure 3-7
Editing a 'SICN' resource

'FONT' resources

The 'FOND' resource is not well described in this preliminary manual. It is intimately related to the 'FONT' resource. Kerning tables and other important information about a font are stored in the 'FOND' that corresponds to the 'FONT'.

For 'FONT' resources, the editing window is divided into four panels: a character editing panel, a sample text panel, a character selection panel, and a typical set of graphics tools. These panels are shown in Figure 3-8.



Beauty is momentary in the mind - / The fitful tracing of a portal / But in the flesh it is immortal.

ASCII	Offset	Width	Location
65	1	16	203

Figure 3-8
Editing a 'FONT' resource

The *character editing panel,* on the left side of the window, shows an enlargement of the selected character. You can edit it, as with FatBits, by clicking bits on and off. Drag the black triangles at the bottom of the character editing panel to set the left and right bounds (that is, the character width). Two of the three triangles at the left side of the panel control the ascent and descent. The third shows the location of the baseline, which is fixed and is displayed only for reference. Below the panel are the ASCII value of the character and the character's offset, width, and location, all in decimal notation.

Warning
Changing the ascent or descent of a character changes the ascent or descent for the entire font.

The *sample text panel,* at the upper right, displays a sample of text in the font currently being edited. (You can change this text by clicking in the text panel and using normal Macintosh editing techniques.)

The *character selection panel* is below the text panel. You can select a character to edit by typing it (using the Shift and Option keys if necessary), or by clicking on it in the row of three characters shown. To move upward through the ASCII range, click on the right character in the row; click on the left character to move downward. The character you select is boxed in the center of the row. (To scroll quickly, click on the right or left character, hold the mouse button down, and move the pointer outside the selection panel, to the right or left.)

The *graphic tools panel,* directly below the character selection panel, offers several familiar graphics-manipulation tools, including the pencil, eraser, circles, and rectangles.

Any changes you make in the character editing panel are reflected in the text panel and the character selection panel.

You can also change the name of a font. The font name is stored in two places: as the name of the 'FOND' resource of that font family, and as the name of the size 0 'FONT' resource. To change the font name, select the individual 'FOND' resource with the name you wish to change, and choose Get Info from the File menu. To maintain consistency, you should also change the name of the 0 point 'FONT' resource; this resource does not show up in the normal display of all fonts in a file. To display it, hold down the Option key while you open the 'FONT' type from the file window. You will see a generic list of fonts. Select the font with the name you wish to change, and choose Get Info.

'PAT ' resources

The 'PAT ' resource editor is shown in Figure 3-9. It displays a single panel, with the pattern shown around a central editing area. This area shows the pattern, enlarged. The outer area shows the pattern at full scale, as you edit the central area as if you were using FatBits.

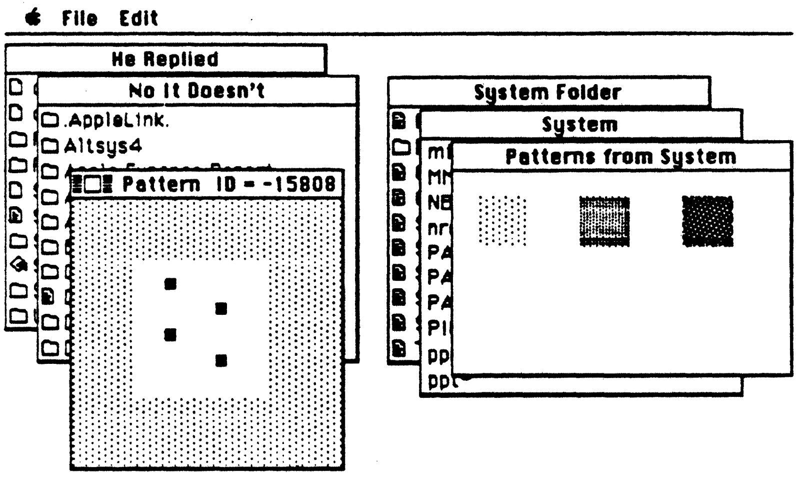


Figure 3-9
Editing a 'PAT ' resource

'PAT#' resources

The 'PAT#' editor is much like the 'SICN' editor, and is shown in Figure 3-10. Instead of displaying a single enlarged picture of the pattern being edited, it shows two. The one on the left is for editing, and is in FatBits; the one on the right shows the resulting pattern at full scale.

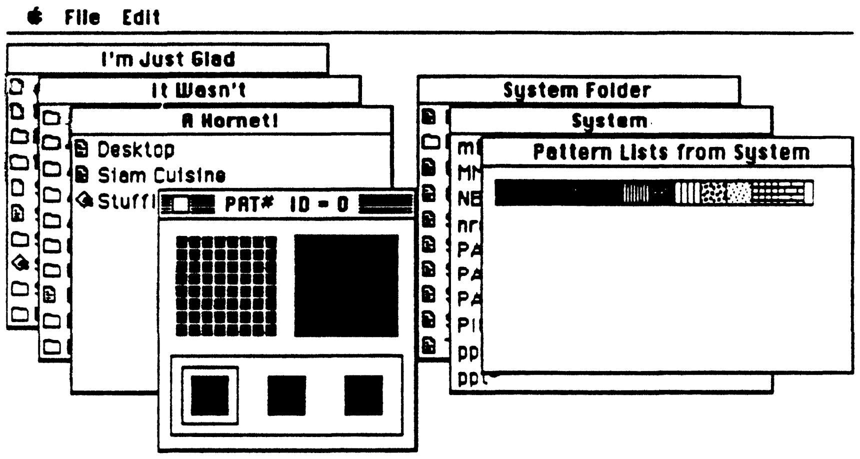


Figure 3-10
Editing a 'PAT#' resource

Hints and kinks

(In the final version of this book, there will be a "ResEdit Hints and Kinks" chapter. For the moment, however, this section is not long enough to take an entire chapter.)

- 'PICT' resource from bitmap. The bitmap image, possibly a screen snapshot or some artwork, is in the data fork of a MacPaint® file (for example), and is not directly accessible to ResEdit. Select and copy (or cut) the part you want. The image is stored in the Clipboard as a 'PICT', which you can paste with ResEdit. For longer-lasting storage, paste it into the Scrapbook, where it is also stored as a 'PICT'.

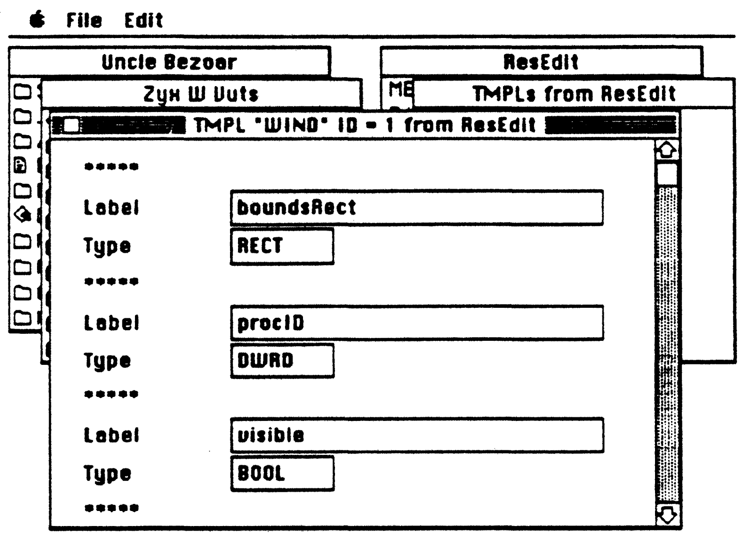
- Pig mode. If you hold down the Command, Option, and Shift keys while selecting About ResEdit from the Apple menu, you can toggle a special stress testing mode. In this mode, ResEdit performs a compact-memory operation and a purge-memory operation each time it receives an event from the queue, excepting null events. This is, clearly, something most people will never have any use for.

- A Suggestion. It is probably a good idea to choose Get Info for ResEdit and increase Application Memory Size to at least 512K. (If you are editing very large resources, even 512K is not sufficient.)

Chapter 4

Making Your Own ResEdit Templates

The generic way of editing a resource is to fill in the fields of a dialog box—for example, this is the way you edit 'FREF', 'BNDL', and 'STR#' resources. The layout of these dialog boxes is set by a template in ResEdit's resource file. The template specifies the format of the resource and also specifies what labels should be put beside the editText items in the dialog box that's used for editing the resource. You can find these templates by opening the ResEdit file and then opening the type window for 'TMPL' resources. For example, if you open the template for 'WIND' resources (the 'TMPL' with name WIND), you'll see the template shown in Figure 4-1.



```
*****
Label    boundsRect
Type     RECT
*****
Label    procID
Type     DWRD
*****
Label    visible
Type     BOOL
*****
```

Figure 4-1
'WIND' template data

The window template consists of the following elements:

1. A RECT (4 words) specifying the boundary of the window.
2. A word that is the procID for the window. (DWRD tells ResEdit to display the word in decimal as opposed to hex.)
3. A Boolean indicating whether or not the window is visible. (BOOL is 2 bytes in the resource but is displayed as a radio button in the dialog window used for editing.)
4. Another Boolean indicating whether or not the window has a close box.
5. A long word that is the reference value (refCon) for the window. (DLNG indicates that it should be displayed in the editor as a decimal number.)
6. A Pascal string (PSTR), the title of the window.

You can look through the other templates and compare them with the structure of those resources to get a feel for how you might define your own resource template. (These templates are equivalent to the resource type declarations contained in the {RIncludes} directory—refer also to the DeRez command in the *MPW Reference,* and the appropriate chapters of *Inside Macintosh.*)

These are the types you have to choose from for your editable data fields:

DBYT, DWRD, DLNG	Decimal byte, word, long word
HBYT, HWRD, HLNG	Hex byte, word, long word
AWRD, ALNG	Word , long align
FBYT, FWRD, FLNG	Byte, word, long fill
HEXD	Hex dump of remaining bytes in resource
PSTR	Pascal string (length byte followed by the characters)
LSTR	Long string (length long followed by the characters)
WSTR	Same as LSTR, but a word rather than a long word
ESTR, OSTR	Pascal string padded to even or odd length (needed for DITL resources)
CSTR	C string (characters followed by a null)
ECST, OCST	Even-padded C string, or odd-padded C string (padded with nulls)
BOOL	Boolean
BBIT	Binary bit
TNAM	Type name (4 characters, like OSType and ResType)
CHAR	A single character
RECT	An 8-byte rectangle
H*nnn*	A 3-digit hex number (where *nnn* < $900); displays *nnn* bytes in hex format

ResEdit does the appropriate type checking for you when you put the editing dialog window away.

The template mechanism is flexible enough to describe a repeating sequence of items within a resource, as in 'STR#', 'DITL', and 'MENU' resources. You can also have repeating sequences within repeating sequences, as in 'BNDL' resources. To terminate a repeating sequence, put the appropriate code in the template as follows.

LSTZ	
...	
LSTE	*List Zero–List End.* Terminated by a 0 byte (as in 'MENU' resources).
ZCNT	
LSTC	
...	
LSTE	*Zero Count/List Count–List End.* Terminated by a zero-based word count that starts the sequence (as in 'DITL' resources).
OCNT	
LSTC	
...	
LSTE	*One Count/List Count–List End.* Terminated by a one-based word count that starts the sequence (as in 'STR#' resources).
LSTB	
...	
LSTE	Ends at the end of the resource. (As in 'acur' and 'APPL' resources.)

The "list-begin" code begins the repeating sequence of items, and the LSTE code is the end. Labels for these codes are usually set to the string "*****". Both of these codes are required.

To create your own template, follow these steps:

1. Open the ResEdit file window.

2. Open the 'TMPL' type window.

3. Choose New from the File menu.

4. Select the list separator (*****) by clicking it with the mouse.

5. Choose New from the File menu. You may now begin entering the *label,type* pairs that define the template. Before closing the template editing window, choose Get Info from the File menu and set the name of the template to the 4-character name of your resource type.

6. Close the ResEdit file window and save changes.

The next time you try to edit or create a resource of this new type, you'll get the dialog box in the format you have specified.

Chapter 5
Extending ResEdit

ResEdit handles all standard resource types (except as noted in Chapter 1). However, you may want to create and edit your own types of resources. You can write extensions to ResEdit in Pascal or C, substituting your own program for parts of its code. This chapter tells you how to do it in Pascal. The final version of this book will contain further information on C language extensions to ResEdit. Some calls have been added to ResEd; there was not sufficient time to document them for this release. They will be included in the final version of this book.

Using ResEd

The program you write must be a Pascal unit or C header file and library. Its interface with ResEdit, if it is written in Pascal, is established by the MPW unit ResEd, contained in the file ResEd.p. Your unit must begin with a `USES` declaration for this unit.

The assembly-language code that "opens up" ResEdit and activates your program is contained in the file ResEd68k.a. It must be linked with your Pascal or C module. When you open a resource of your type, ResEdit will jump to this code.

To install your new editor after it is compiled, place it in ResEdit's file (using ResEdit itself) with the type 'RSSC' and a unique ID number. (You can use ResEdit to check the ResEdit file to see which ID numbers are already taken.) Your editor's name in the ResEdit file must be of the form @ABCD, where ABCD is the name you have assigned to the new type it edits. Install your picker (also of type 'RSSC') with the name ABCD (without the commercial "at" sign.).

Pickers and editors

When ResEdit uses your program, it looks for two general capabilities: a picker and an editor. Pickers and editors are separate from the main code of ResEdit and hence may be supplied by user-written software.

The *picker* is the part that displays all the resources of your type in the resource type window. It is given the resource type and should display all resources of that type in the current resource file, using a suitable display format. If the picker is given an open call and there's a suitable editor, it should launch the editor.

The *editor* is the code that displays and lets you edit a particular resource. The editor is given a handle to the resource object and should open an edit window (or windows) for you.

Note that pickers and editors can be opened from anywhere. For instance, a dialog editor might open an icon picker so that you could choose an appropriate icon. While using the icon picker, you could open the icon editor if you wanted to create a new icon.

Writing a ResEdit extension in Pascal

A sample ResEdit extension program is given in the file "Res*XXXX*Ed.p". In this sample, *XXXX* represents your resource type.

The sample program is called by means of the `EditBirth` routine when a resource of type *XXXX* must be edited. This routine is passed two handles: a handle to the resource to be edited (the same handle that would be received by using a `GetResource` call) and a handle back to the picker that has launched this editor. The `EditBirth` routine and other necessary routines will be described in more detail in the full release of this reference.

The program then creates a window and sets up any data structures needed to operate. Because it may be loaded in and out of memory during any given session and because it doesn't have access to global variables, it creates a handle to a data structure to hold all data that needs to be preserved between calls. It stores the handle in the edit data structure `r`*XXXX*`rec`. Note that the handle to the edit data structure is stored in the window's `refCon` parameter. ResEdit uses this to identify which subprogram is to receive a given call.

ResEdit will determine which editor should receive which events, so you need to do very little event decoding in your editor. During an update event, the `BeginUpdate` and `EndUpdate` calls are done by ResEdit, not by the extension program.

Here are two things to remember when writing a ResEdit extension:

- ☐ Always know which resource you are requesting and where it will come from. Many resource files may be open at any given time. Whenever a resource is needed, make sure which resource file you are accessing by using `UseResFile` or similar operations.
- ☐ Your editor may be called with an empty handle in order to create an entirely new instance of the type you edit.

The ResEd interface

The ResEd unit contains data structures, procedures, and functions that you can access from your extension program. They are described in the remainder of this chapter.

Data structures

The ResEd unit declares the data structures described in this section, which provide communication between extension programs and ResEdit.

The resource map entry

```
ResMapEntry = RECORD
                rID: integer;
                rNameOff: integer;
                rLocn: longint;
                rHndl: Handle
              END; {ResMapEntry}
```

The resource map entry record accesses the contents of a resource map directly. It is used by the routines `Get1MapEntry` and `Get1IMapEntry`, described later in this chapter.

The parent record

Each picker or editor has its own object handle. This handle must start with a handle to its parent's object, followed by the name distinguishing the father. This name will be part of the son's window title. The next field should be the window of the object, which may be used by the son to get back to the father, through the `refCon` field in the `windowRec` record. The rest of the handle can be of any format.

```
ParentPtr = ^ParentRec;
ParentHandle = ^ParentPtr;
ParentRec = RECORD
             father: ParentHandle;
             name: Str64;
             wind: WindowPeek;
             rebuild: boolean    {flag set by son to indicate}
                                 {that world has changed so}
                                 {father should rebuild list}
           END;          {ParentRec}
```

The picker record

The standard picker record has this structure:

```
PickPtr = ^PickRec;                   {Any type is OK here.}
PickHandle = ^PickPtr;
PickRec = RECORD
           father: ParentHandle;      {back ptr to dad}
           fName: Str64;
           wind: WindowPtr;           {picker window}
           rebuild: boolean;
           pickID: integer;           {resource ID of picker}
           rType: ResType;            {type for picker}
           rNum: integer;             {resfile number}
           rSize: longint;            {size of a null resource}
           nInsts: integer;           {number of instances}
           instances: ListHandle;     {list of instances}
           drawProc: Ptr;             {list draw procedure}
           scroll: ControlHandle      {scroll bar}
          END; {PickRec}
```

Launcher routines

The `GiveEBirth` procedure is used to launch your editor.

The GiveEBirth procedure

`PROCEDURE GiveEBirth (`*b*`: Handle;` *pick*`: PickHandle);`

`GiveEBirth` launches an editor for a specific resource. The parameter *b* is the handle to that resource as returned by `GetResource`; *pick* is the handle to your own picker record.

There are four other routines, `CallPBirth` and `CallEBirth`, `CallEvent`, and `CallMenu`, that are included in the ResEd unit, the use of which is not recommended unless your editor invokes another editor (as, for example, the DLOG editor invokes the DITL editor).

Information-passing routines

The four routines described in this section are used to pass information between ResEdit and your program. `CallEvent` and `CallMenu` are rarely used.

The CallInfoUpdate procedure

```
PROCEDURE CallInfoUpdate(oldID, newID: integer; object: longint; id: integer);
```

`CallInfoUpdate` tells the picker that launched your editor that an ID or name has been changed, added, or deleted, making it necessary to rebuild the picker list. The parameter *object* identifies the resource and editor; *id* identifies the resource ID of the object.

The PassMenu procedure

```
PROCEDURE PassMenu(menu, item: integer; father: ParentHandle);
```

`PassMenu` passes menu selections on to any son pickers or editors that you have launched with `ShowInfo`. Here is an example of its use:

```
PassMenu(file, close, myObj);    {Tell all subsidiary windows to close.}
```

Window utilities

The five ResEdit window-handling routines described in this section are available to your program.

The WindAlloc function

```
FUNCTION WindAlloc: WindowPtr;
```

`WindAlloc` returns a pointer to a window record to be used by your editor. Using this routine instead of allocating your own window pointer can help reduce heap fragmentation. `WindAlloc` has no parameters.

The WindFree procedure

```
PROCEDURE WindFree(w: WindowPtr);
```

`WindFree` releases the window pointer allocated by `WindAlloc`. Use this when you terminate your editor and you are finished with its window, but only if you used `WindowAlloc` to create the window.

The WindList function

```
FUNCTION WindList(w: WindowPtr; nAcross: integer; pt: Point; drawProc: integer):
ListHandle;
```

`WindList` creates a new empty list and returns a handle to that list. For more information on lists and a description of the `WindList` parameters, see the List Manager chapter in *Inside Macintosh,* Volume IV.

The WindOrigin procedure

```
PROCEDURE WindOrigin(w: WindowPtr);
```

`WindOrigin` moves the window pointed to by *w* to a position down and to the right of the current front window.

The WindSetup function

```
FUNCTION WindSetup(width, height: integer; t, s: Str255): WindowPtr;
```

`WindSetup` creates and automatically positions a new window with the given *width* and *height* and displays a title formed by `ConcatStr(t, s)`. It returns a pointer to the window.

Resource utilities

The five ResEd routines listed in Table 5-1 act the same as the Macintosh Resource Manager routines, but operate only on the currently selected resource file. For further information about these routines and their parameters, see *Inside Macintosh,* Volume IV, Chapter 3. Six additional resource utility routines available in ResEd are described below.

Table 5-1
Resource Manager calls

ResEd routine		Corresponding Macintosh routine
FUNCTION	Count1Type	Count1Types
PROCEDURE	Get1IndxType	Get1IndType
FUNCTION	Count1Res	Count1Resources
FUNCTION	Get1Index	Get1IndResource
FUNCTION	Get1Res	Get1Resource

The CurrentRes function

```
FUNCTION CurrentRes: integer;
```

`CurrentRes` returns the ID number of the currently selected resource file.

The Get1MapEntry procedure

PROCEDURE Get1MapEntry(VAR *theEntry*: ResMapEntry; *t*: ResType; *id*: integer);

`Get1MapEntry` accesses the resource map for a resource of type *t* and ID number *id*, placing the result in *theEntry*. For a description of resource maps, see "Format of a Resource File" in *Inside Macintosh*, Volume I, Chapter 5.

The Get1IMapEntry procedure

PROCEDURE Get1IMapEntry(VAR *theEntry*: ResMapEntry; *t*: Restype; *index*: integer);

`Get1IMapEntry` acts like `Get1MapEntry`, described above, except that it refers to its resource by index instead of ID number.

The GetResLoad function

```
FUNCTION GetResLoad: boolean;
```

`GetResLoad` returns the current value set by the Macintosh procedure `SetResLoad`.

The RevertResource function

FUNCTION RevertResource(*h*: Handle): boolean;

`RevertResource` restores a resource being edited to the state it was in before editing started. The parameter *h* is a handle to the resource. `RevertResource` returns a value of `false` if the resource was newly added by ResEdit (and, therefore, no longer exists after the reversion), `true` otherwise.

Miscellaneous utilities

The ResEd routines described in this section perform miscellaneous tasks in a ResEdit extension program. They are listed in alphabetical order.

The AbleMenu procedure

`PROCEDURE AbleMenu(`*`menu`*`: integer; `*`enable`*`: longint);`

`AbleMenu` enables and disables menu items. The parameter ***menu*** is a menu ID; ***enable*** is a mask. `AbleMenu` differs from the Resource Manager routines `EnableItem` and `DisableItem` in that it acts on the entire menu. (Values for the mask have changed recently, and are available in ResEd.)

The AddNewRes function

`FUNCTION AddNewRes(`*`hNew`*`: Handle; `*`t`*`: ResType; `*`idNew`*`: integer; `*`s`*`: Str255): boolean;`

`AddNewRes` has the same parameters and performs the same action as the Macintosh procedure `AddResource`, with the addition that it returns `true` if it is successful and `false` otherwise. For information about `AddResource`, see *Inside Macintosh,* Volume I, Chapter 5.

The AppRes procedure

`PROCEDURE AppRes;`

`AppRes` performs the action of `UseResFile` on ResEdit itself. It is useful if you need to get a resource from the resource editor, such as an `ICON` or `DITL`, for your editor to use.

The BubbleUp procedure

`PROCEDURE BubbleUp(`*`h`*`: Handle);`

`BubbleUp` sets up the correct heap zone, and then performs the Memory Manager routine `MoveHHi`. For information about `MoveHHi`, see *Inside Macintosh,* Volume II, Chapter 1.

The BuildType function

`FUNCTION BuildType(`*`t`*`: ResType; `*`l`*`: ListHandle): integer;`

Given a list that has been initialized with no rows, `BuildType` builds a list of all resources of type *t* from the current resource file. (See the `WindList` routine described above.) If `SetResLoad(true)` has been called, all the resources will be loaded in also. `BuildType` returns a count of the number of instances currently in the list.

The ClearHand procedure

`PROCEDURE ClearHand(`*`h`*`: Handle);`

`ClearHand` clears the resource referenced by the handle *h* to all zeros.

The ConcatStr procedure

`PROCEDURE ConcatStr(VAR` *`str1`*`: Str255;` *`str2`*`: Str255);`

`ConcatStr` concatenates *str2* to *str1*, leaving the result in *str1*.

Warning
This routine does not check for aggregate string length in excess of 255 characters. Please be careful!

The CopyRes function

`FUNCTION CopyRes(VAR` *`h`*`: Handle;` *`makeID`*`: boolean;` *`refNum`*`: integer): Handle;`

Given a handle *h* to a resource, `CopyRes` makes a copy of the resource to the resource file specified by *refNum*. Note that the handle is changed, so you can't keep track of your resource by saving its handle before using `CopyRes`. If *makeID* is `true`, a unique ID will be assigned to the copy; otherwise, it retains the ID of the original. `CopyRes` returns a handle to the new copy (in the new file).

The DoListEvt procedure

`PROCEDURE DoListEvt(`*`e`*`: EventRecord;` *`l`*`: ListHandle);`

Given an event *e* and a list *l*, `DoListEvt` does the standard dispatch to the List Manager. (See *Inside Macintosh*, Volume IV.) The port must be set to the window that owns the event. `DoListEvt` also enables the File menu and draws controls in the window.

The DupPick function

`FUNCTION DupPick(`*`h`*`: Handle;` *`c`*`: cell;` *`pick`*`: PickHandle): Handle;`

`DupPick` duplicates a resource referenced by *h*, adds it to the picker list referenced by *pick*, and performs an `InvalRect` action on *thePort* for the new cell *c*. It also makes the new cell the selection.

The ErrorCheck function

`FUNCTION ErrorCheck(`*`err`*`,` *`msgID`*`: integer): boolean;`

Given a result code *err* and a message ID number *msgID*, `ErrorCheck` brings up an error dialog box if *err* is nonzero. If *msgID* is negative, the box displays a fatal error message retrieved from `STR#` resource ID 128; otherwise, it displays a message retrieved from `STR#` resource ID 129. `ErrorCheck` returns `true` if successful, `false` otherwise. When creating new strings for use by `ErrorCheck`, be sure to add them to the end of the existing list in the `STR#` resource.

The FileNewType function

`FUNCTION FileNewType(`*`types`*`: ListHandle; VAR` *`s`*`: Str255): boolean;`

FileNewType puts up a dialog box containing a list of the types of resources that can be edited. The list is referenced by *types*. FileNewType then returns true if the user selects a type, false if Cancel is clicked. The selected type is returned in *s*.

The FixHand procedure

```
PROCEDURE FixHand(s: longint; h: Handle);
```

FixHand makes sure the object to which *h* is a handle is *s* bytes long. If it's longer, FixHand shrinks it; if it's shorter, FixHand expands it and fills the extension with zeros.

The HandleCheck function

```
FUNCTION HandleCheck(h: Handle; msgID: integer): boolean;
```

HandleCheck checks to see if the handle *h* is nil or empty. If it is either, HandleCheck returns false; otherwise, it returns true. The parameter *msgID* is the ID from STR# resource ID 129. If this function fails, it calls ErrorCheck.

The MetaKeys procedure

```
PROCEDURE MetaKeys(VAR cmd, shift, opt: boolean);
```

MetaKeys accesses the modifier flags for the last event, returning their values in *cmd, shift,* and *opt*. For further information about modifier flags, see *Inside Macintosh,* Volume I, Chapter 8.

The NewRes function

```
FUNCTION NewRes(s: longint; t: ResType; l: ListHandle; VAR n: integer): Handle;
```

Given a size *s*, NewRes allocates a new handle, clears it, adds it to the current resource file as a resource of type *t*, adds it to the list *l*, and returns a handle to the new resource. The parameter *n* is the item number in the list *l*. If this function fails, it returns a nil handle.

The PickEvent procedure

```
PROCEDURE PickEvent(VAR evt: EventRecord; pick: PickHandle);
```

PickEvent handles an event contained in *evt* for a standard picker referenced by *pick*. Call PickEvent from your picker's event procedure.

The PickInfoUp procedure

```
PROCEDURE PickInfoUp(oldID, newID: integer; pick: PickHandle);
```

PickInfoUp handles an information update (identified by *oldID* and *newID*) for a standard picker referenced by *pick*. Call PickInfoUp from your picker's DoInfoUpdate procedure. If the ID has not changed, oldID and newID will, of course, be identical.

The PickMenu procedure

```
PROCEDURE PickMenu(menu, item: integer; pick: PickHandle);
```

`PickMenu` handles a menu selection (identified by *menu* and *item*) for a standard picker referenced by *pick*. Call `PickMenu` from your picker's `DoMenu` procedure.

The RCalcMask procedure

```
PROCEDURE RCalcMask(srcPtr, dstPtr: Ptr; srcRow, dstRow, height, words: integer);
```

`RCalcMask` calculates a mask for the given source bit image and puts it into the destination bit image. The parameters *srcPtr* and *dstPtr* reference the source and destination bit images; *srcRow, dstRow, height,* and *words* define the area on which `RCalcMask` operates. For further information about bit images, see *Inside Macintosh,* Volume I, Chapter 6.

The ResEdID function

```
FUNCTION ResEdID: integer;
```

`ResEdID` returns the resource ID of the editor or picker that calls it.

The ResEditRes function

```
FUNCTION ResEditRes: integer;
```

`ResEditRes` returns the resource ID of the resource editor.

The ResEverest procedure

```
PROCEDURE ResEverest;
```

`ResEverest` sets the current resource file to the last one opened.

The RSeedFill procedure

```
PROCEDURE RSeedFill(srcPtr, dstPtr: Ptr; srcRow, dstRow, height, words: integer; seedH, seedV:
integer);
```

Given a source bit image, a destination bit image, and a seed location, `RSeedFill` fills the bits in the destination with black. The parameters *srcPtr* and *dstPtr* reference the source and destination bit images; *srcRow, dstRow, height,* and *words* define the area on which `RSeedFill` operates; *seedH* and *seedV* define the seed location. For further information about bit images, see *Inside Macintosh,* Volume I, Chapter 6.

The ScrapCopy procedure

```
PROCEDURE ScrapCopy(VAR h: Handle);
```

`ScrapCopy` copies the ResEdit scrap into the resource identified by *h*.

The ScrapEmpty procedure

```
PROCEDURE ScrapEmpty;
```

`ScrapEmpty` empties the ResEdit scrap. Call it before doing a Cut or Copy operation.

The ScrapPaste procedure

```
PROCEDURE ScrapPaste(resFile: integer);
```

`ScrapPaste` pastes the resources from the ResEdit scrap to the file identified by the ID number *resFile*.

The SetETitle procedure

```
PROCEDURE SetETitle(h: Handle; VAR str: Str255);
```

Given a handle *h* to a resource, `SetETitle` concatenates the resource's ID with its name and places the result into *str*.

The SetResChanged procedure

```
PROCEDURE SetResChanged(h: Handle);
```

`SetResChanged` marks the resource *h* as changed so that it will be updated when your program terminates.

The ShowInfo procedure

```
PROCEDURE ShowInfo(h: Handle; dad: ParentHandle);
```

`ShowInfo` puts up a GetInfo window for the resource referenced by *h* that belongs to the father object referenced by *dad*.

Index

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh® computers and Microsoft® Word. Proof pages were created on the Apple LaserWriter® Plus. POSTSCRIPT®, the LaserWriter page-description language, was developed by Adobe Systems Incorporated.

Text type is ITC Garamond® (a downloadable font distributed by Adobe Systems). Display type is ITC Avant Garde Gothic® Bullets are ITC Zapf Dingbats® Some elements, such as program listings, are set in Apple Courier, a fixed-width font.

2/21/88

ERS

MPW LL(1) PARSE TABLE GENERATOR [LL1GENERATOR]

Development Systems Group

Version 0.3 -- November 1987

Modification History

0.00	9/9/87	Eagle I. Berns	-First Specification release
0.1	9/22/87	Eagle I. Berns	-Removed restriction to read in Terminal, non Terminal, and action names. They are now embedded in production syntax. -Added ability to insert initialization data/programs between %{ line and %} line. Allowed names to be up to 31 chars. -Allowed comments between /* and */. Generalized form of input for productions. -Allowed ∂ (option-d) as an escape character. -Fixed bugs.
0.2	10/9/87	Eagle I. Berns	-Added '-C' option to LL to produce C-type inclusion files for drivers. -Included a C equivalent version of the existing pascal sample driver. -Added initialization capability through the "OTHERWISE/default" capability within the actions. -Added %{token ... facility to allow assignment by user of terminal token values. -Changed the naming convention on the included/generated files. -Fixed problems arising from "end" conditions (such as at End-of-line, EOF, etc.)
0.3	11/2/87	Eagle I. Berns	-Added -d options to create ll.definitions file with #defines or CONST symbols for terminal productions - for use with users own parser.

-Allow terminal definitions without numeric value equates so quotes don't have to be used in the production definitions for terminals.

-Converted error message form to look like pascal error messages and allow clicking on error message file/line to go direct to problem.

-Changed tool name to LL1PARSERGENERAOR

-fixed general bugs.

Project Identification

Product Name

MPW LL(1) Grammar Parser - LL1GENERATOR

Related Documents

Compiler Construction, Waite/Goos, Springer-Verlang, 1984.

Syntax of Programming Languages, Theory and Practice, Roland C. Backhouse, 1979.

Compiler Construction: Theory and Practice, William A. Barrett & John D. Couch, SRA, 1979.

Compilers, Principles, Techniques, and Tools (the "dragon book"), Aho, Sethi, & Ullman, Addison Wesley, 1986.

The Design and Construction of Compilers, Robin Hunter, John Wiley & Sons, 1981.

Compiler Design Theory, (The systems programming series), P.M. Lewis II, D.J. Rosenkrantz, R.E. Stearns, Addison Wesley, 1976.

Product Abstract

The MPW LL(1) Grammar Parser is composed of two distinct parts: The LL1PARSER-GENERATOR tool, and the PGP or PGC tool. The LL1GENERATOR tool takes in the definition of a language which must be a well defined LL(1) Grammar, along with any embedded actions, and optionally the source code for those actions, and produces a series of three files containing the parsing tables, the action code processor, and the initialization constants. A script which is included with the product, merges these files, and compiles a version of the PGP or PGC tool, which is tailored to the original input grammar. This newly created tool may be invoked from MPW, the input will be "executed" by the tool, and any pre-specified actions will be performed. A sample grammar is included to show a straight forward application of the tool to write a simple calculator tool. The references listed above provide more than adequate description of the definition of an LL(1) grammar.

User Environment

These tools will run on a Mac+, Mac SE, or MAC II, in the MPW shell environment.

File Identification

There are a number of files generated and/or used by this product. Many can be used without involvement by the programmer, although all are accessible to them, including the source of the driver program (but not the source of the LL(1) grammar table generator). These files, their types, and use are described in the following table.

file name	File type	Usage
LL1GENERATOR	MPW tool	This is the tool which creates the actual tables.
LL.RESOURCE	MPW rez input	LL1GENERATOR creates this file which is suitable for input to REZ and contains all tables that are needed by the driver program.
LL.INITIALIZE	MPW Source text file	LL1GENERATOR creates this file which contains the constants which need to be initialized by the driver.
LL.ACTIONS	MPW Source text file	LL1GENERATOR creates this file which contains the routine invoked when an action was specified. The routine contains the users on action source.
LL.DEFINITIONS	MPW Source text file	LL1GENERATOR creates this file which contains a file of CONST or #define constants when the -d option is used. The items included are the internal terminal symbol id's or those which the user has specified in the "%{token" segment of the source (see below).

PGP.P-PGC.C	MPW Source text file	This is the source used as the skeleton for the inclusion of the files created above. The script below compiles this, and places the results in a user file. One is Pascal source, the other is the equivalent code written in C.
PGP.R-PGC.R	MPW Rez file	This contains the resources used by PGP.P and PGC.C
PGM	MPW make file	This file will create the end user tool by compiling, and linking the source of PGP.P, or PGC.C and the user files.
filename.X	MPW tool	This is the final user MPW tool. It's arguments/usage are explained below.

Program Usage

LL1GENERATOR --- LL(1) Grammar Parser

Syntax LL1GENERATOR [*option*] *filename*

Description LL1GENERATOR processes a source file (*filename*) containing the definitions for grammar presented to it. The form of the input is described below.

Type Tool.

Input The Input file *filename* contains the source for the grammar. The data may begin immediately with the productions. The start symbol must be the Left-Hand-Side of the first production. (The form of the productions is shown below. Alternatively, you may begin the input with a line containing **%{** . If you do, all lines read until a line containing **%}** is found will be copied into the LL.INITIALIZE file mentioned above. After the productions, a line with **%** must be placed, if the source for the actions to be performed follows.

The Source of actions are "CASE" statement alternatives for PASCAL, or "switch" statement alternatives for C. (See below the example for PASCAL) LL1GENERATOR will surround the code with the appropriate CASE/SWITCH statement and procedure body needed for use in the PGP and PGC statement. Examining the LL.ACTIONS file once you've executed the LL1GENERATOR command will show what is produced.

An additional type of input input is available: the **%{TOKEN ... %}** input. The data between these lines must be composed of triplets of texts in the following form: A terminal symbol used in the grammar, one or more blanks; an optional separator symbol of some form (usually "EQU" or "="); one or more blanks; and finally a numeric value. For example:

```
+  EQU  8
-  EQU  9
++ EQU  18
...
```

If this form of the input precedes the grammar definition, it will have two effects. First, It will not be necessary to surround the terminal symbols with the single quote character ('). And second, and this is primary purpose of the option, all output tables generated will use the numeric values specified rather than the sequential numbering scheme normally generated by LL1GENERATOR. Identification numbers generated internally for the non-terminals will begin with a numeric value one higher than the largest value specified in the section above. **NOTE:** if this option is used, it is assumed that the user will have their own driver program for processing terminal symbols - the PGP and PGC programs are not designed to use this mode. (although, a few simple modifications could be made to do so). This option is primarily provided for those who may already have their own scanner defined. One additional warning: If the tokens are pre-defined with the above mechanism, any terminal symbols other than those which have been defined will be considered to be an error, and the program will terminate. An optional form of token input is to NOT specify anything but a list of the terminal names to be used. The user will thus not be required to quote the items in the productions, and this system will assign its own internal numeric values. Examples of the various forms are shown in the section describing the LL.DEFINITIONS file itself.

Output The LL.INITIALIZE, LL.RESOURCE, LL.ACTIONS, and LL.DEFINITIONS files described above are generated.

Diagnostics Errors are written to diagnostic output.

Status The following status values may be returned:

0	No Error
1	Syntax Error

Options

-p : Write progress information to diagnostic output.
-c : Generate output files for inclusion in a "C" driver vs. Pascal.
-d : Generate output file LL.DEFINITIONS with the appropriate form for constants that were specified as terminals with the internal id's specified by the system or those generated by the "%{token" field

Example LL1GENERATOR CALCP.LL {see the complete example described below}

Limitations LL1GENERATOR will ONLY work on well formed LL(1) Grammars.

COMPLETE SAMPLE PROGRAM

The file **CALCP.LL** contains the following data:

```
%{
        VAR
                Stack:                          ARRAY[0..500] OF integer;
                tos:                                    INTEGER;
%}

%{tokens
        $EOL
        +
        -
        *
        /
        (
        )
        $VAR
        $NUMBER
%}

P                       :       P2 P1 ;

P1              :       P2 P1 ;
P2              :       E {pulnsho} '$EOL' ;
E   :  T E1 ;
E1              :       + T {pulnadd} E1
                        |       - T {pulnsub} E1 ;
T   :  F T1 ;
T1              :       * F {pulnmlt} T1
                        |       / F {pulndiv} T1 ;
F                       :       '(' E ')'
                        |       '$VAR'
                        |       '$NUMBER' {pshnum} ;
/* these null non terminals are needed */
P1              : ;
T1              : ;
E1              : ;
%
```

```
PshNum:
        BEGIN
                tos:=tos+1;
                stack[tos]:=NumSymbol; {"Numsymbol" is defined in the Drivers
                                        GetNextSymbol Routine}
        END;
PulNAdd:
        BEGIN
                stack[tos-1]:=stack[tos-1]+stack[tos];
                tos:=tos-1;
        END;
PulNSub:
        BEGIN
                stack[tos-1]:=stack[tos-1]-stack[tos];
                tos:=tos-1;
        END;
PulNMlt:
        BEGIN
                stack[tos-1]:=stack[tos-1]*stack[tos];
                tos:=tos-1;
        END;
PulNDiv:
        BEGIN
                stack[tos-1]:=stack[tos-1] DIV stack[tos];
                tos:=tos-1;
        END;
PulNSho:
        BEGIN
                Writeln(stack[tos]:0);
                tos:=tos-1;
        END;
OTHERWISE tos:=0; {init stage does this}
```

A Few things that are important to notice in the above input data are the following: Terminal signals are denoted by enclosing them in single quote marks; Actions are identified by surrounding them with curly braces; Comments may go anywhere; the ∂ (option-d) symbol may be used to allow the following character to stand for itself (to allow for example, single quotes and curly braces in the grammar); Multiple LHS declarations may use the OR (|) symbol to avoided repeating its name; Productions are terminated by a semi-colon (;) ; Finally, notice that there are some special terminal symbols $EOL, $NUMBER, and $VAR. These symbols are recognized as special by the parser included in the sample (obviously, the person designing the GETNEXTSYMBOL routine may, or may not choose to use this). $EOL is generated whenever there is an End Of Line read on input of the string to be parsed. $NUMBER will "chew-up" a sequence of digits from 0 to 9 and make one integer value from them. $VAR will "chew-up" a series of letters and/or numbers which start with a letter. These can be omitted if the user wishes and they may use actual letter or number terminals. Since the source of PGP and PGC is included, a user may add additional special types, and handle them in the "GETNEXTSYMBOL" routine.

When LL1GENERATOR is run with the above input data, the following output is generated in the files specified:

LL.INITIALIZE

(Note that some of the output shown below is produced by the LL1GENERATOR program itself.)

```
        VAR
                Stack:                      ARRAY[0..500] OF integer;
                tos:                            INTEGER;
CONST
 pulnsho             = -100;
 pulnadd             = -101;
 pulnsub             = -102;
 pulnmlt             = -103;
 pulndiv             = -104;
 pshnum              = -105;

 NTF = 10;
 TL  = 9;
 NTL = 17;
```

LL.RESOURCE

```
Resource 'PROD' (1000) {
 {
/* 0 */ {11,12},
/* 1 */ {11,12},
/* 2 */ {13,-100,1},
/* 3 */ {14,15},
/* 4 */ {2,14,-101,15},
/* 5 */ {3,14,-102,15},
/* 6 */ {16,17},
/* 7 */ {4,16,-103,17},
/* 8 */ {5,16,-104,17},
/* 9 */ {6,13,7},
/* 10 */ {8},
/* 11 */ {9,-105},
/* 12 */ {},
/* 13 */ {},
/* 14 */ {}
 }
};

Resource 'GIDX' (1000) {
 {
1,4,7,11,14,19,24,27,32,37,41,43,46,47,48
 }
};

Resource 'TERM' (1000) {
 {
/* 0 */ "EOF                ",
/* 1 */ "$EOL               ",
/* 2 */ "+                  ",
/* 3 */ "-                  ",
/* 4 */ "*                  ",
/* 5 */ "/                  ",
/* 6 */ "(                  ",
/* 7 */ ")                  ",
/* 8 */ "$VAR               ",
/* 9 */ "$NUMBER            "
 }
};

Resource 'NTRM' (1000) {
 {
/* 10 */ "P                 ",
/* 11 */ "P2                ",
/* 12 */ "P1                ",
/* 13 */ "E                 ",
/* 14 */ "T                 ",
/* 15 */ "E1                ",
/* 16 */ "F                 ",
/* 17 */ "T1                "
 }
};
```

```
Resource 'ACTN' (1000) {
 {
/* -100 */ "pulnsho                 ",
/* -101 */ "pulnadd                 ",
/* -102 */ "pulnsub                 ",
/* -103 */ "pulnmlt                 ",
/* -104 */ "pulndiv                 ",
/* -105 */ "pshnum                  "
 }
};

Resource 'PNAM' (1000) {
 {
/* 0 */ "P                 ",
/* 1 */ "P1                ",
/* 2 */ "P2                ",
/* 3 */ "E                 ",
/* 4 */ "E1                ",
/* 5 */ "E1                ",
/* 6 */ "T                 ",
/* 7 */ "T1                ",
/* 8 */ "T1                ",
/* 9 */ "F                 ",
/* 10 */ "F                ",
/* 11 */ "F                ",
/* 12 */ "P1               ",
/* 13 */ "T1               ",
/* 14 */ "E1               "
 }
};

Resource 'GRAM' (1000) {
   /* Triplets: row,column,production no.      */
   /* Grammar->M[row,column] := production no.; */
 {
  10,6,0,
  10,8,0,
  10,9,0,
  11,6,2,
  11,8,2,
  11,9,2,
  12,0,12,
  12,6,1,
  12,8,1,
  12,9,1,
  13,6,3,
  13,8,3,
  13,9,3,
  14,6,6,
  14,8,6,
  14,9,6,
  15,1,14,
  15,7,14,
  15,3,5,
  15,2,4,
  16,9,11,
  16,8,10,
  16,6,9,
  17,1,13,
  17,2,13,
  17,3,13,
  17,7,13,
  17,5,8,
  17,4,7
 }
};
```

LL.ACTIONS

```
PROCEDURE ActionProc(actionNumber:integer);
 BEGIN
  CASE ActionNumber OF
PshNum:
        BEGIN
                tos:=tos+1;
                stack[tos]:=NumSymbol; {"Numsymbol" is defined in the Drivers
                                        GetNextSymbol Routine}
        END;
PulNAdd:
        BEGIN
                stack[tos-1]:=stack[tos-1]+stack[tos];
                tos:=tos-1;
        END;
```

```
PulNSub:
        BEGIN
                stack[tos-1]:=stack[tos-1]-stack[tos];
                tos:=tos-1;
        END;
PulNMlt:
        BEGIN
                stack[tos-1]:=stack[tos-1]*stack[tos];
                tos:=tos-1;
        END;
PulNDiv:
        BEGIN
                stack[tos-1]:=stack[tos-1] DIV stack[tos];
                tos:=tos-1;
        END;
PulNSho:
        BEGIN
                Writeln(stack[tos]:0);
                tos:=tos-1;
        END;
OTHERWISE tos:=0;{init stage does this}
 END;
 END;
```

LL.DEFINITIONS

If the -d options had been specified, this file would have been created and would have contained the following:

```
CONST
 $EOL = 1;
 + = 2;
 - = 3;
 * = 4;
 / = 5;
 ( = 6;
 ) = 7;
 $VAR = 8;
 $NUMBER = 9;
 EOF = 0;
```

Note that this makes no sense for pascal, nor would it for the "C" version, and would not be specified if the terminal symbols were constants such as those shown.
But had the %{token field been used with data such as:

```
%{token
        $EOL         equ 1
        PLUS         equ 2
        MINUS        equ 3
        TIMES        equ 4
        DIV          equ 5
        LPAREN       equ 6
        RPAREN       equ 7
        $VAR         equ 8
        $NUMBER      equ 9
%}
```

... the CONST section would have been as follows:

```
CONST
        $EOL         = 1;
        PLUS         = 2;
        MINUS        = 3;
        TIMES        = 4;
        DIV          = 5;
        LPAREN       = 6;
        RPAREN       = 7;
        $VAR         = 8;
        $NUMBER      = 9;
        EOF          = 0;
```

(of course, this assumes that these symbols were then used in the grammar, such as PLUS instead of '+', etc.)

The same result would have occurred had the use NOT placed any "equ" values in the token section (as shown below) since this is the sequence that LL1GENERATOR would have generated internally. i.e.

```
%{token
        $EOL
        PLUS
        MINUS
        TIMES
        DIV
        LPAREN
        RPAREN
        $VAR
        $NUMBER
%}
```

Had the user inserted their own equ id values they, of course, would have been the ones used for LL.DEFINITIONS.

Once this portion of the process is completed, the PGP or PGC program, under control of the Make file PGM can be executed. This Make file simple does a PASCAL or C compile, link, and rez, to generate an MPW tool with the name CALCP.LL.X. The form of the PGM call is **PGM*filename*** and additionally the letter **C** if a C form of generation is required. Once completed, the tool created may be called in the following manner:

CALCP.LL.X [-P] [*filename*]

The -P puts out progress information during the parsing of each input line, which may come either from the console, or from the file *filename*. An example of what is produced from CALCP.LL.X is shown below (although it is quite trivial).

```
CALCP.LL.X
1+2*3-4*5+6*(8-5) <enter>
5
3+4*5 <enter>
23
<command-.>

all done.
```

The file CALCP.LL is available to test the working of the two tools. Keep in mind, that you may change PGP.P and PGC.C as much (or as little) as you need to accommodate the parsing needs of your particular grammar. Usually, this takes the form of replacing the GETNEXTSYMBOL routine, and/or adding additional constants, and variables used by the grammars actions.

Macintosh® MacsBug Reference

APDA Final Draft
11 July 1988
Developer Products Publications

2/21/88

Contents

Preface

About This Manual

Welcome to MacsBug 6.0. MacsBug has been around for a long time. If you're already a MacsBug user, you'll be glad to know that it's new and improved. Not only does it have lots of new features, it's been completely rewritten so that it makes more sense and can grow. A summary of what's new and what's changed is given in Appendix E: you'll probably want to start there.

MacsBug is Apple's assembly-language debugger for Macintosh programmers. If you have written, or are trying to write, a program for the Macintosh, you'll find MacsBug a powerful debugger with many unique capabilities. If you aren't actually writing a program, but have a good basic understanding of *Inside Macintosh*, you'll find MacsBug a helpful tool for learning more about the Macintosh. (In fact, MacsBug was used frequently in the writing of *Inside Macintosh* to determine how particular routines actually worked.)

Chapter 1 provides an overview of MacsBug. This includes a description of the hardware and software configurations MacsBug works with, what kind of debugger MacsBug is, and the files on the MacsBug disk.

Chapter 2 introduces the MacsBug commands and how they fit into various debugging strategies.

Chapter 3 provides a complete specification of the MacsBug command language, including command syntax, operation, and examples.

Appendix A contains a summary of all MacsBug commands.

Appendix B list the error messages returned by MacsBug.

Appendix C describes MacsBug internals for advanced programmers.

Appendix D details how you can call MacsBug from within your program.

Appendix E outlines the differences between MacsBug version 5.5 and MacsBug 6.0.

Appendix F provides tips, shortcuts, and interesting facts about MacsBug.

Appendix G covers procedure name definition for advanced programmers.

Notation conventions

The following notation conventions are used to describe MacsBug commands:

literal	Plain text indicates a word that must appear in the command exactly as shown. Special symbols (-, §, &, and so on) must also be entered exactly as shown.
variable	Items in italics can be replaced by anything that matches their definition.
[optional]	Square brackets mean that the enclosed elements are optional.

repeated...	An ellipsis (...) indicates that the preceding item can be repeated one or more times.
either I or	A vertical bar (I) indicates an either/or choice.
(grouping)	Parentheses indicate grouping and are used when both items of a choice can be specified and repeated; that is, (param1 I param2 ...).
{ Return }	Brackets are used in examples to indicate that the specified key should be pressed. They are also used to enclose comments.

Command names and filenames are not sensitive to case.

Chapter 1
MacsBug Overview

About MacsBug

MacsBug is a Motorola 68000-family assembly-language debugger customized for the entire Macintosh® family of computers. First introduced in 1981, MacsBug has continued to evolve along with the Macintosh. Version 6.0 represents a new generation for MacsBug. While compatibility with earlier versions has been maintained—virtually all of the old commands are supported—MacsBug has been completely rewritten to take full advantage of the power and sophistication of the new machines.

MacsBug 6.0 runs on the Macintosh Plus, Macintosh SE, and Macintosh II, and is designed to support future members of the 68000 family. It handles the MC68881 floating-point coprocessor and the MC68851 Memory Management Unit (MMU). It also supports external displays on the Macintosh Plus and Macintosh SE, as well as various screen sizes and bit depths on Macintosh II displays. There's no need to customize MacsBug for particular configurations since it determines the attributes of the machine at system startup.

MacsBug 6.0 works with all versions of Macintosh system software, and is compatible with MultiFinder™.

MacsBug 6.0 does not work with the 64K ROMs, nor does it run on the Macintosh XL.

Macintosh debugging

MacsBug uses as little of the Macintosh system software as possible. This lets systems programmers debug their software without having to worry about the debugger using the code they're debugging. But MacsBug isn't only a systems-level debugger. The high degree of interaction between a Macintosh application and the system also makes MacsBug a powerful tool for debugging applications.

MacsBug is an assembly-language debugger. If you're writing programs in a high-level language like C or Pascal, you'll more often want to use the Symbolic Application Debugging Environment (SADE™). SADE lets you debug your program at the source-code level, which means you don't need to know assembly language or map object code back to your program's source-level instructions. If you need to, SADE lets you monitor program execution at the machine level as well.

SADE does have its limitations, however, and high-level programmers will find that MacsBug picks up where SADE leaves off. Specifically:

- SADE uses the Macintosh system software extensively, and in the case of a severe crash may not be operable. MacsBug lets you examine the remains to try to determine what went wrong.

- If RAM is severely limited, you may not be able to run SADE. MacsBug is lean and mean.

MacsBug is loaded at system startup and sits quietly in RAM until it's invoked. Unlike debuggers that expect a target program to work with, MacsBug lets you look at practically anything running on the Macintosh—toolbox and operating system routines, applications, desk accessories, and so on.

You can suspend program execution at any point, either manually (by pressing the interrupt switch or a key that you define) or programmatically (by calling special traps from within your program). And since MacsBug needs so little of the system to operate, it can be used even in the case of fatal system errors. Whenever the System Error Handler is called, or when a 68000 exception occurs, MacsBug takes control and lets you look around.

Once MacsBug has been invoked, you can enter commands to

- Display and set memory and registers.
- Disassemble memory.
- Set execution breakpoints.
- Step and trace through both RAM and ROM.
- Monitor system traps.
- Display and check the system and application heaps.

The next chapter introduces the MacsBug features and how they fit into various debugging strategies. Chapter 3 provides a complete specification of the MacsBug command language, including command syntax, operation, and examples.

MacsBug files

MacsBug is shipped on its own 3.5-inch disk. Included on this disk are files than can be used with the Macintosh Programmer's Workshop (MPW™) and with ResEdit™ to customize MacsBug. Details are provided in later chapters, and each file includes specific instructions for creating your own MacsBug resources.

Chapter 2
Debugging With MacsBug

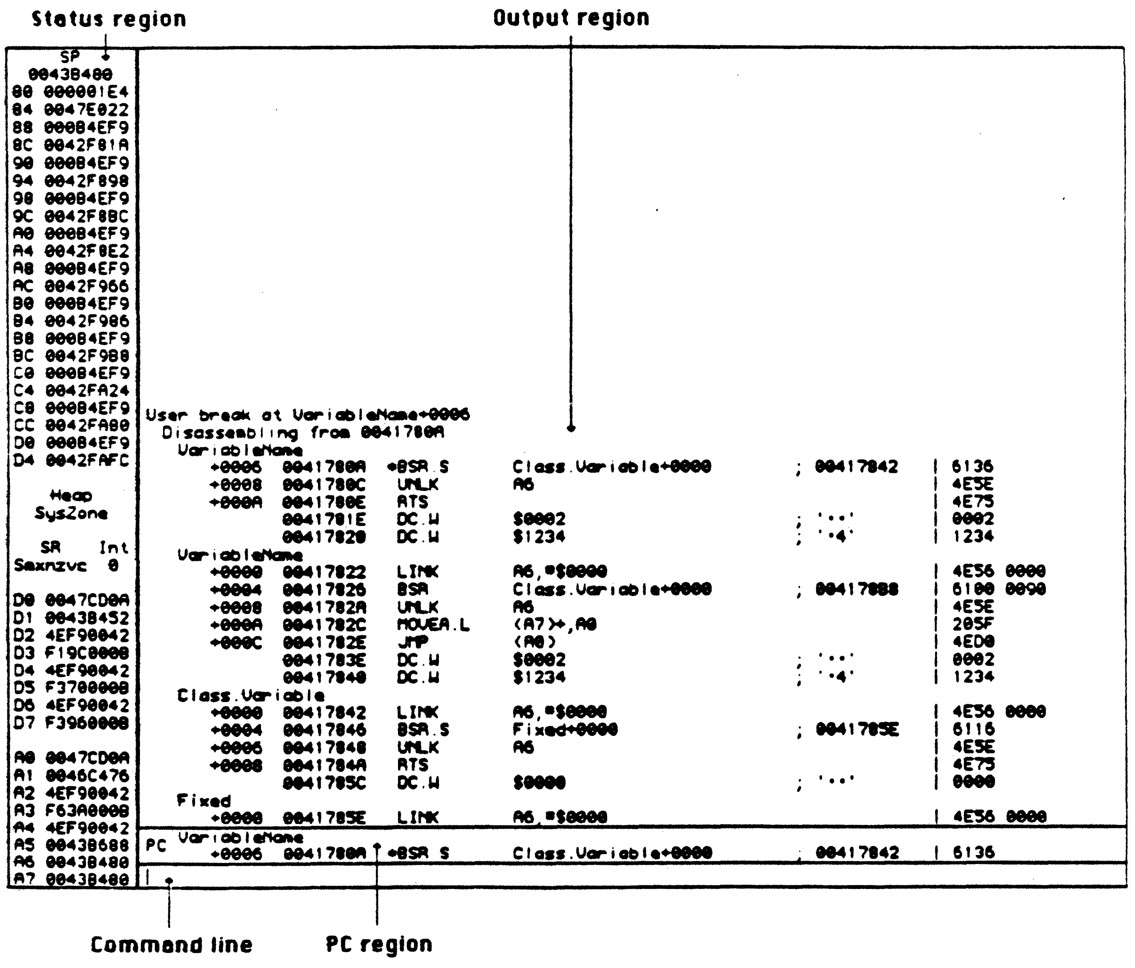
Getting started

MacsBug is installed at system startup and resides in RAM until shutdown. In order to be recognized at boot time, the MacsBug file must be in the System Folder on the startup disk.

The system startup code identifies the MacsBug file by name. To prevent MacsBug installation indefinitely, you can rename the MacsBug file, or move the file from the System Folder. To override MacsBug installation for a single session only, simply hold down the mouse button during startup.

After a successful installation, the message "MacsBug installed" is displayed below the "Welcome to Macintosh" message. The startup application (typically the Finder™) is then launched.

The simplest way to invoke MacsBug is by pressing the interrupt switch; this generates an NMI exception and suspends program execution. MacsBug takes control and displays a screen like that shown in Figure 1.



```
    SP
 0043B480
00 000001E4
04 0047E022
08 00084EF9
0C 0042F81A
10 00084EF9
14 0042F898
18 00084EF9
1C 0042F8BC
A0 00084EF9
A4 0042F8E2
A8 00084EF9
AC 0042F966
B0 00084EF9
B4 0042F986
B8 00084EF9
BC 0042F988
C0 00084EF9
C4 0042FA24
C8 00084EF9
CC 0042FA80
D0 00084EF9
D4 0042FAFC

   Heap
 SysZone

  SR    Int
Smxnzvc  0

D0 0047CD0A
D1 0043B452
D2 4EF90042
D3 F19C0008
D4 4EF90042
D5 F3700008
D6 4EF90042
D7 F3960008

A0 0047CD0A
A1 0046C476
A2 4EF90042
A3 F63A0008
A4 4EF90042
A5 0043B688
A6 0043B480
A7 0043B480
```

```
User break at VariableName+0006
 Disassembling from 0041780A
  VariableName
    +0006  0041780A  •BSR.S     Class.Variable+0000     ; 00417842   | 6136
    +0008  0041780C   UNLK      A6                                   | 4E5E
    +000A  0041780E   RTS                                            | 4E75
           0041781E   DC.W      $0002                   ; '••'       | 0002
           00417820   DC.W      $1234                   ; '•4'       | 1234
  VariableName
    +0000  00417822   LINK      A6,#$0000                            | 4E56 0000
    +0004  00417826   BSR       Class.Variable+0000     ; 004178B8   | 6100 0090
    +0008  0041782A   UNLK      A6                                   | 4E5E
    +000A  0041782C   MOVEA.L   (A7)+,A0                             | 205F
    +000C  0041782E   JMP       (A0)                                 | 4ED0
           0041783E   DC.W      $0002                   ; '••'       | 0002
           00417840   DC.W      $1234                   ; '•4'       | 1234
  Class.Variable
    +0000  00417842   LINK      A6,#$0000                            | 4E56 0000
    +0004  00417846   BSR.S     Fixed+0000              ; 0041785E   | 6116
    +0006  00417848   UNLK      A6                                   | 4E5E
    +0008  0041784A   RTS                                            | 4E75
           0041785C   DC.W      $0000                   ; '••'       | 0000
  Fixed
    +0000  0041785E   LINK      A6,#$0000                            | 4E56 0000
PC VariableName
    +0006  0041780A  •BSR S     Class.Variable+0000     ; 00417842   | 6136
|
```

Figure 1. MacsBug Display

Note: Another way to invoke MacsBug is to define an 'FKEY' resource containing the two instructions needed—Debugger ($A9FF) and RTS ($4E75).

To see the application screen again, press the tilde key (~) or the escape key (esc). To return to the MacsBug display, press any character key.

If you have multiple screens, MacsBug uses the "Welcome to Macintosh" screen by default. You'll probably want your application on the larger screen and MacsBug on the smaller screen. To select a different screen for the MacsBug display, press the Option key while clicking on the Monitor icon from the Control Panel and then drag the Macintosh icon to the desired screen.

At the bottom of the MacsBug display is the command line, indicated by a flashing bar cursor. MacsBug accepts the standard editing keys (Delete, Left Arrow, Right Arrow), as well as several special functions:

Command-Left Arrow	Move cursor left one word.
Command-Right Arrow	Move cursor right one word.
Command-Delete	Delete the word to the left of the cursor.
Command-V	Restore previous command line(s) for editing.

Multiple commands, separated by semicolons, can be entered on the command line. To execute the command(s) on the command line, press Return or Enter. Pressing Return without entering a command repeats the last command.

The Return key can be also used as a toggle to pause and resume the execution of a command. To cancel the execution of a command, press any other key. (Note, however, that execution cannot then be resumed.)

Thorough on-line help information that includes the syntax of all commands can be displayed with the HELP command.

The largest area of the screen is the output region. MacsBug output falls into three categories, indicated by three levels of indentation:

- The reason for the break. MacsBug tells which 68000 exception, Macintosh system error, or user-specified break caused MacsBug to be invoked.

- Messages. For each command you enter, MacsBug gives a message either confirming execution or explaining a failure.

- Command output.

Output scrolls up (and eventually off) the screen as new commands are executed. The LOG command lets you save all MacsBug output, to either a file or an ImageWriter® printer.

Immediately above the command line is the PC (program counter) region; it shows the address of the next instruction to be executed, along with the disassembly of that instruction. In the case of a fatal error, it shows the last instruction executed (in other words, the instruction that caused the crash).

The area on the left side of the screen, known as the status region, displays information about the system. At the top is the address contained in the stack pointer (register A7), followed by the bytes at the top of the stack. The number of bytes displayed varies with the screen size and the format of the display. The SHOW command lets you specify the display in word, long word, and ASCII format; it also lets you specify another area of memory for display.

Below the stack data is the name of the current heap; by default it's the application heap. You can change the current heap with the HX (Heap Exchange) command. The HZ (Heap Zones) command tells you all known heap zones, and works with MultiFinder.

The rest of the status region shows the contents of the CPU registers. Several commands give additional register information. The TF (Total Floating-Point) and TM (Total MMU) commands show the contents of the floating-point and MMU registers respectively. The TD (Total Display) command displays the CPU registers in the output region. Since the CPU registers are constantly updated and displayed in the status region, the TD command is useful for remembering register values between commands.

There are several ways to leave MacsBug. The simplest way is with the G (Go) command; program execution resumes at the current program counter. If MacsBug was invoked due to an unexpected error condition, it may not be possible to resume program execution. Depending on the severity of the error condition, it may be necessary to relaunch the application (EA command), relaunch the shell (ES command), restart the system (RS command), or reboot the machine (RB command).

Specifying things

Most of the MacsBug operations—setting breakpoints, displaying memory, disassembling code—need an actual address to work with. To make life easier, MacsBug provides a number of different alternatives to specifying hard addresses.

Whenever possible, MacsBug accepts and returns symbols in place of addresses. Procedure names are the most common example of this. Most compilers for the Macintosh have the option of embedding character names after the code generated for each procedure or function. (Compiler writers will want to see Appendix G for details on procedure name definition.) If you've used this option, you can specify a procedure name and offset whenever MacsBug wants an address. Conversely, MacsBug returns addresses as offsets from procedures whenever it can. For instance, if the instruction shown in the PC is part of a valid procedure, the PC window gives the name and offset of that instruction.

Several commands work specifically with symbols. The WH (Where) command provides mapping between symbols and addresses. When given an address or a symbol name, MacsBug gives you the other item. You can display a list of all procedures known to MacsBug with the SD (Symbol Display) command. You can also use the SD command to display a list of trap names. You can disassemble any of your application's procedures with the IR command.

To translate a symbol name into an address, MacsBug must search the current heap. Since this search process can be slow, MacsBug provides the SX (Symbol Exchange) command for disabling the use of symbol names.

> Note: Advanced programmers may find themselves dealing with multiple files (code segments, for instance) having the same symbol names. The RN (Resource Number) command lets you restrict symbol matching to a file with a given reference number.

MacsBug also allows the creation of macros. Macros are simple text string substitutions, and can be used to create command name aliases, reference global variables, and name common expressions. Macros are expanded before the command line is executed, and can thus contain anything you can type in a command line.

You can create macros on the fly with the MC command, or include them in a resource file. (See the MC command for details.) The MCD command lists the macros known to MacsBug, and the MCC command clears one or all macros.

How did I get here?

When your program crashes unexpectedly, you'll start with several clues. MacsBug tells you what 68000 exception or system error ID caused the crash. The PC area gives you the instruction that caused the crash. The location of the instruction, whether in ROM or at an offset from a procedure, is also given. You can examine the code immediately preceding the crash by using the IP command.

One approach is to examine the stack for the procedure call history. If your program uses the LINK A6 procedure prolog, the SC6 command returns the calling history. If it doesn't, or if you are in a part of the ROM that doesn't use A6 links, you'll need to use the SC7 command. This command finds possible return addresses on the stack. You can use these addresses to examine the stack yourself. You can also use the addresses in other MacsBug commands.. Be aware, though, that the SC7 command will almost certainly include old or invalid values (in other words, addresses not in the current calling chain), since local stack variables can change the stack top without changing the contents.

Another way to find out where a program has been is by recording the A-trap calls it makes, using the ATR (A-Trap Record) command. When recording has been turned on, MacsBug records all trap calls in a circular buffer. When the buffer is full, the oldest calls will be overwritten by new calls. You can define the size of the buffer and thereby the number of traps recorded. (See the ATR command for details.) You may want to consider always enabling trap recording; the performance cost isn't very great. To see the information recorded, use the ATP (A-Trap Playback) command.

In the same way that trap recording lets you build a trap history in a buffer, the ATT command lets you direct that history to the screen or to a log file. Tracing to the screen is useful if you have two screens. MacsBug can take over one screen and display the history as your program executes on the other screen. In cases where the program crashes so badly that MacsBug cannot be invoked, you'll still have a trap history available.

Enabling logging with the LOG command and tracing to a file is useful if you want to record a large number of calls and can't afford to dedicate the memory for the trap recording buffer. Another benefit of log files is that you can use your editor to help examine the data.

Controlling program execution

MacsBug provides a set of commands that let you control and watch the execution of your program. Two commands let you execute instructions one at a time. The S (Step) command executes a single 68000 instruction, stops at the next instruction, and returns to MacsBug. The contents of the program counter—in other words, the next instruction to be executed—are disassembled and displayed. You can also step through a specified number of instructions, or until a condition is met (for instance, when a register contains a particular value).

When the S command reaches a subroutine or an A-trap call, it steps right in. Particularly with ROM routines, which are often very long and typically not of interest, you'll probably want to use the SO (Step Over) command instead. The SO command works exactly like the S command except that it treats A-trap calls and subroutines as a single instruction, stopping at the first instruction after the A-trap or subroutine returns. (With traps having the auto-pop bit set, MacsBug returns to the address on the top of the stack at the time of the trap call.)

While stepping through code, MacsBug decodes conditional statements (DBcc, Bcc, and Scc instructions) to determine whether branches will be taken or will fall through. This information is shown to the right of the PC information.

If you've stepped into a procedure with the S command and want to get out, you can use the MR (Magic Return) to move to the end of the procedure. The MR command needs to know where the return address is; for this reason, it's a good idea to use the LINK A6 prolog for your procedures.

If you're stepping through your program and find you want to move past some code, you can use the GT (Go Till) command to resume execution until a specified address is reached.

Stopping at a particular place

Once you've narrowed down the location of a bug, you may want MacsBug to stop when a particular point in your program is reached. There are several ways of doing this.

The ATB (A-Trap Break) command lets you specify a break when A-traps are encountered. You can specify individual traps or a range of traps, as well as conditions that must be met. For instance, you could specify a break when the HFSDispatch trap is encountered and the value of register D0 is 6 (which is the routine selector for the DirCreate routine). You can also specify commands to be executed once MacsBug has been invoked, making life a little easier.

Another way to stop program execution is to set a breakpoint at a specified address, using the BR command. The address can be given as an actual address, or as an offset from a procedure name. This information will have been found by disassembling or stepping through your code. The BR command also lets you specify commands to be executed when the breakpoint is reached. You can specify multiple breakpoints; MacsBug stores this information in a table, which you can see at any time with the BRD command. Breakpoints remain set until you clear them with the BRC command.

The BR command can be useful in working with A-traps as well as with your own code. With some ROM routines, the actual trap is often preceded by glue code that sets up the parameters. Whereas the ATB command stops right before the trap is made, the BR command can be used to stop at the point where your program calls the routine, letting you examine what goes on with the glue code.

An advantage of using breakpoints is that they don't require changes to your source code, and can be used after the application has been built. However, breakpoints cannot be set in a procedure until the segment containing that procedure is loaded and the address determined. One way around this problem is to specify a break from within your procedure by using the traps Debugger ($A9FF) and DebugStr ($ABFF). Debugger is a system trap that invokes MacsBug and displays the message "User break at <addr>." DebugStr additionally lets you supply a custom message for display, as well as MacsBug commands for execution. (For a description of how to declare and use these traps, see Appendix D.)

The DX (Debugger Exchange) command lets you disable breaks from the Debugger and DebugStr traps without having to go in and remove them from your program.

Watching for memory to change

Several commands let you determine when and where a particular area of memory is being changed. One common problem is when a program inadvertently changes the contents of a memory location. You can detect when a range of memory changes by using the SS (Step Spy) command. This command checksums a given range and then executes instructions one at a time until the checksum changes. The SS command can slow down a program considerably, so MacsBug treats a long word as a special case and optimizes for speed. If you suspect a certain range of memory is being altered, you usually don't need to check the whole range but can check just a long word within the range. If you must check a long range, you'll probably want to use a hardware emulator. (You can also use the SS command as a way of slowing down certain routines, those that draw to the screen, for instance, so you can actually watch how they work.)

A variation on the SS command, the ATSS (A-Trap Step Spy) command lets you checksum a memory range before specified A-traps are executed.

The CS (Checksum) command lets you monitor whether a range of memory has changed. The first time you execute the CS command, you specify a range and MacsBug computes a checksum. Subsequent CS commands compute the checksum and compare it with the previous value.

Displaying and setting memory

The DB, DW, DL, and DP commands display respectively a byte, word, long word, and page (128 bytes) of memory. With the DM (Display Memory) command, you can specify a number of bytes to be displayed. Often you'll want to look at the contents of a data structure consisting of fields of various different sizes. The DM command lets you specify templates for displaying memory in a structured format.

The TMP command lists the names of all templates known to MacsBug. See the description of this command for instructions on defining your own templates.

The SB, SW, and SL commands let you set bytes, words, and long words in memory. The SM command lets you assign values of varying size; the size of the assignment is determined by the value.

Checking the heap

Several commands let you examine and monitor heap zones. The HD (Heap Dump) command displays information about all blocks in the current heap, as well as a summary of the heap allocation. If you want only the summary information, you can use the HT (Heap Totals) command.

One of the more common bugs is dereferencing a handle to a block that has moved, potentially corrupting the heap. Two commands are useful in detecting this problem. The HC (Heap Check) command checks the current heap and reports any errors. If the problem is reproducible, the ATHC (A-Trap Heap Check) command can be used to check the heap before trap calls.

Exercising your program

It's possible to simulate a worst-case memory situation to exercise your application. The HS (Heap Scramble) command moves all relocatable blocks whenever they *might* be moved, in other words, whenever the NewPtr, NewHandle, ReallocHandle, SetPtrSize, or SetHandleSize traps are called. (With SetPtrSize and SetHandleSize, the heap is scrambled only if the block size is being increased.)

The DSC command turns on the Extended Discipline™ utility. This program examines parameters before A-traps are called and checks results after the calls complete. If Extended Discipline detects an error, MacsBug is invoked. (See the Extended Discipline manual for details.)

The dot address

MacsBug provides a way of saving and specifying addresses between successive commands; it's so useful that it deserves a separate section.

MacsBug maintains a variable, known as "dot," that contains the last address of interest from certain commands. The "." character refers to this address and can be used in any command that expects an address. The commands that set the "dot address" are ones that are often followed by another command using the same address.

Dot is used primarily as a shorthand notation between one command and the next. For instance, you might type `WH name` to find a particular procedure. The WH command sets dot to the address returned, letting you then type `IL .` to disassemble code, or `BR .` to set a breakpoint at the start of the procedure.

Dot can also be used as a placeholder. For instance, the DM command displays memory starting from a specified address and sets dot to that address. You can resume execution, reenter MacsBug later, and type `DM .` to display the same memory. This technique is useful for watching for an area of memory to change.

The commands that set dot are as follows:

- The commands for displaying memory—DM, DP, DB, DW, and DL—all set dot to the address of the first byte displayed.
- The commands for setting memory—SM, SB, SW, and SL—set dot to the address of the first byte changed. These commands also set the last command to the DM command. This means that after setting memory you can simply press Return to display the memory just set.
- The WH command sets dot to the address of the procedure or trap located.
- The F command sets dot to the first byte of the string that was found.
- The IL, IP, and ID commands set dot to the address of the first instruction disassembled.
- The dot address is also used in connection with heap commands. Any command that scans the heap—HD, HC, and SD, for instance—can receive a heap error. If the error concerns a particular block (as opposed to the entire heap), MacsBug sets dot to the address of the block header. Typing `DM .` will display the block in question. MacsBug also scans resource maps while examining resource blocks in the heap. Resource map errors set dot to the address of the bad map.

Chapter 3
MacsBug Commands

Command syntax

MacsBug commands have the following format:

COMMAND required parameters [optional parameters]

Parameters can be numbers, text literals, symbols, or expressions combining these elements. MacsBug provides full command line evaluation, so any parameter can be entered as an expression. The general form of an expression is:

value1 [operator value2]

Parentheses can be used to control the order of evaluation. Expressions always evaluate to a 32-bit value unless `.W` or `.B` follows the specified value (in which case the word or byte is sign-extended to 32 bits). Expressions can evaluate to either a numeric or a Boolean value depending on the operators used. The operation of certain commands varies depending on the type of expression. For instance, the BR command will break after *n* times if the given expression is numeric, or when a certain condition is met if the expression is Boolean.

Values

Depending on the command, there are a variety of different ways to specify values:

registers	All 68000-family registers use their Motorola names. MMU 64-bit registers and floating-point registers are not allowed in expressions.
numbers	Numbers are hexadecimal by default, but can be preceded by the "$" character in the case of conflicts with registers An and Dn. Numbers are decimal if preceded by the "#" character. The unary operators "+" and "–" can precede any number, but must appear before the "#" or "$" characters.
symbols	Symbols are found by searching the heap, and evaluate to an address. (See Appendix G for details on procedure name definition.) Partial name matching is supported. If you enter `SD My`, for instance, all symbols that start with "My" are shown.
traps	A-traps are specified by trap number in the range A000 to ABFF or by trap name. Trap names can be preceded by the "†" character (Option-T) in the case of conflicts with symbol names.
strings	Strings are sequences of characters surrounded by single (') or double (") quotation marks. There is no padding to word or long word boundaries; each character in the string is 1 byte.

.	The "." character specifies the dot address; see Chapter 2 for a detailed discussion on using this character.
:	The ":" character indicates the address of the start of the procedure shown in the program counter window. This character is not valid if no procedure name exists for PC.

Operators

These are the operators allowed in expressions, in order of precedence:

(	)	Grouping
@ (prefix)	^ (postfix)	Address indirection
!	NOT	Bitwise or Boolean NOT
*		Multiplication
/	MOD	Division
+		Addition
–		Subtraction
=	==	Equal
<>	!=	Not equal
<		Less than
<=		Less than or equal
>		Greater than
>=		Greater than or equal
&	AND	Bitwise or Boolean AND
|	OR	Bitwise or Boolean OR
XOR		Bitwise or Boolean XOR

Note: `@addr` is the same as `addr^`. `Addr^.B` or `addr^.W` fetch only a byte or word from *addr*; the value is then sign-extended to 32 bits.

Command descriptions

This section contains descriptions of all MacsBug commands, arranged alphabetically. For each command, the parameters are given and the operation of the command discussed. Where appropriate, examples are provided. A list of the entire set of commands can be found in Appendix A.

ATB — A-Trap Break

Syntax ATB[A] [*trap* [*trap*]] [*n* | *expr*] [';*cmds*']

Description The ATB command sets a breakpoint at the specified A-trap(s). Traps can be specified by either trap number or trap name. Appending the letter *A* to the ATB command tells MacsBug to break only when the given trap is called from the application heap. (Note that this means the current application heap at the time the ATB command was entered.) Specifying two traps indicates a range of traps; MacsBug breaks at every trap encountered *within* this range. If no traps are specified, a default range of A000 through ABFF is used.

If *n* is specified, MacsBug breaks only after a given trap has been encountered *n* times. If *expr* is specified, MacsBug breaks only when a given trap has been encountered and *expr* is TRUE. If neither *n* nor *expr* is given, MacsBug breaks each time the trap is encountered. You can also supply one or more commands to be executed once the break conditions are satisfied; each command must be preceded by a semicolon and enclosed in quotation marks.

You can set multiple trap breaks with different break conditions or commands. MacsBug checks the table until an entry satisfies the break conditions. The break commands for this entry are executed. Later entries in the table (that also satisfy the break conditions) are ignored.

Be aware that MacsBug stores the information for breakpoints, step commands, and A-trap commands in a single table. New entries are entered at the end of the table. It's possible to receive the error message "Entry will not fit in the table" while entering an ATB command if step commands, BR commands, and other A-trap commands have already filled this table.

Examples

```
ATB                             {break on all traps}
ATB GetNextEvent                {break on GetNextEvent trap}
ATB A000 A010                   {break on traps Open through Allocate}
ATB HFSDispatch D0.W = 6        {break on HFSDispatch when register D0=6 (DirCreate)}
ATB SizeWindow  ';DM (SP+6)^ WindowRecord'  {break on SizeWindow, then display }
                                            { from the contents of the long word }
                                            { 6 bytes above the stack pointer }
                                            { using the WindowRecord template}
```

For a display of the trap table after having set these actions, see the ATD command.

See also ATC, ATD

ATC — A-Trap Clear

Syntax ATC [*trap* [*trap*]]

Description The ATC command clears all actions on the specified traps; in other words, it cancels the ATB, ATT, ATHC, and ATSS commands. Traps can be specified by either trap number or trap name. Specifying two traps indicates a range of traps; MacsBug cancels actions for all traps within this range. If no traps are specified, all trap actions for all traps are cleared.

The ATC command comes in handy when you want to set an action for most, but not all, of the traps in a particular range. For instance, you may think you want to break at all toolbox traps but soon find that you can do without a break at every call to GetNextEvent. One way around this is to set two ranges around GetNextEvent with the ATB command. An easier way is to set the action on the whole range and use the ATC command to exclude the GetNextEvent trap. Be aware, however, that MacsBug accomplishes this by doing the dirty work for you, itself setting two new ranges around GetNextEvent. This means that, even though you are ostensibly clearing a trap action, you are actually creating an additional entry in the A-trap table, and could conceivably receive the error message "Entry will not fit in the table."

Example

Assume the trap table (displayed by using the ATD command) looks like this:

```
A-Trap actions from System or Application
 Trap Range  Action     Cur/Max or Expression       Commands
 A000  ABFF  Break      00000000 / 00000001
```

After you enter the command:

```
ATC  GetNextEvent
```

the trap table looks like this:

```
A-Trap actions from System or Application
 Trap Range  Action     Cur/Max or Expression       Commands
 A000  A96F  Break      00000000 / 00000001
 A971  ABFF  Break      00000000 / 00000001
```

See also ATB, ATD, ATHC, ATSS, ATT

ATD — A-Trap Display

Syntax ATD

Description The ATD command displays the A-trap table(s), which list all actions set with the ATB, ATT, ATHC, and ATSS commands. Two A-trap tables may be displayed, depending on which actions have been set. One table lists all actions restricted to the application heap (using the A parameter), and another lists actions that apply to either the system heap or the application heap.

Both tables have the same format. The trap range for the action is shown in the first column and the type of action is shown in the second column. If a number of iterations (*n*) was specified with the action, it's shown in the third column, preceded by the actual number of iterations so far. If a logical expression was entered instead, it's shown in the third column. The fourth column shows any commands that were specified for execution upon breaking into MacsBug.

Example

In this example, the following trap breaks were set previously with the ATB command:

```
ATB
ATB GetNextEvent
ATB A000 A010
ATB HFSDispatch D0.W = 6
ATB SizeWindow   ';DM (SP+6)^ WindowRecord'
```

The trap table displayed by the ATD command is given below. Note that traps are represented by trap number; you can determine the corresponding trap name by using the WH command.

```
ATD
 A-Trap actions from System or Application
  Trap Range   Action     Cur/Max or Expression        Commands
  A000  ABFF   Break       00000000 / 00000001
     A970      Break       00000000 / 00000001
  A000  A010   Break       00000000 / 00000001
     A060      Break      D0.W = 6
     A91D      Break       00000000 / 00000001          ;DM (SP+6)^ WindowRecord
```

See also ATB, ATC, ATHC, ATSS, ATT, WH

ATHC — A-Trap Heap Check

Syntax ATHC[A] [*trap* [*trap*]] [*n* | *expr*]

Description The ATHC command checks the consistency of the heap before executing the specified A-trap(s). If the heap is found to have been corrupted, MacsBug is invoked and an error is displayed; see the HC command for a list of possible errors.

Traps can be specified by either trap number or trap name. Appending the letter *A* to the ATHC command tells MacsBug to check only when the given trap is called from the application heap. (Note that this means the current application heap at the time the ATHC command was entered.) Specifying two traps indicates a range of traps; MacsBug checks for every trap encountered within this range. If no traps are specified, a default range of A000 through ABFF is used.

If *n* is specified, MacsBug checks only after a given trap has been encountered *n* times. If *expr* is specified, MacsBug checks only when a given trap has been encountered and *expr* is TRUE. If neither *n* nor *expr* is given, MacsBug checks each time the trap is encountered.

See also ATC, ATD, HC

ATP— A-Trap Playback

Syntax ATP

Description The ATP command plays back the information saved while trap recording is on. (For details on trap recording, see the ATR command.) This information includes the trap name and the contents of the program counter (PC). For operating-system traps, the value of registers A0 and D0 are shown, as well as the 8 bytes pointed to by register A0. For toolbox traps, ATP shows the value of register A7 and the 8 bytes to which it points.

Example

In the example below, SetPort is the most-recently executed trap.

```
ATP
 Trap calls in the order in which they occurred
  A030 OSEventAvail
    PC = 004C7346
    A0 = 004871F8  003A 1F34 2000 004B   D0 = 0000FFFF
  A970 GetNextEvent
    PC = 004C2BCA
    A7 = 0048724C  0048 7290 FFFF 0020
  A030 OSEventAvail
    PC = 004C7346
    A0 = 004871DC  004C 16B4 004D 013C   D0 = 00000000
  A031 GetOSEvent
    PC = 004C7334
    A0 = 00487290  0000 0000 0000 000A   D0 = 00000000
  A9B4 SystemTask
    PC = 004C2800
    A7 = 004871F0  0000 FFFF 0048 7290
  A874 GetPort
    PC = 40815150
    A7 = 004871C4  0048 71C8 0048 7290
  A924 FrontWindow
    PC = 40815154
    A7 = 004871C4  0000 0000 003A 1D40
  A873 SetPort
    PC = 408151AE
    A7 = 004871C8  003A 1D40 004D 013C
```

See also ATR

ATR— A-Trap Record

Syntax

ATR[A] [ON | OFF]

Description

The ATR command turns trap recording on and off; if no parameter is passed, the command toggles between modes. Trap recording saves information about the *n* most recently executed traps. By default, MacsBug records the last eight traps. You can, however, specify any number by using the resource type 'mxbr' with a resource ID of 100. The format of this resource is simply an Integer. Since the traps are saved in a circular buffer, space is the only penalty for recording more than eight traps; time is not a factor.

Appending the letter *A* to the ATR command tells MacsBug to record information only for traps called from the application heap. (Note that this means the current application heap at the time the ATR command was entered.)

The information saved, which can be displayed with the ATP command, includes the trap name and the contents of the program counter (PC). For operating-system traps, the value of registers A0 and D0 are saved, as well as the eight bytes pointed to by register A0. For toolbox traps, ATR saves the value of register A7 and the 8 bytes to which it points.

See also

ATP

ATSS — A-Trap Step Spy

Syntax ATSS[A] [*trap* [*trap*]] [*n* | *expr*], *addr1*[*addr2*]

Description The ATSS command calculates the checksum for the given memory range before executing the specified A-trap(s). If the checksum value changes, MacsBug is invoked. If *addr2* is omitted, ATSS waits for the long word at *addr1* to change. The ATSS command is optimized for speed with a long word; longer checksum ranges can be slow.

Traps can be specified by either trap number or trap name. Appending the letter *A* to the ATSS command tells MacsBug to check only when the given trap is called from the application heap. (Note that this means the current application heap at the time the ATSS command was entered.) Specifying two traps indicates a range of traps; MacsBug checks for every trap encountered within this range. If no traps are specified, a default range of A000 through ABFF is used.

If *n* is specified, MacsBug checks only after a given trap has been encountered *n* times. If *expr* is specified, MacsBug checks only when a given trap has been encountered and *expr* is TRUE. If neither *n* nor *expr* is given, MacsBug checks each time the trap is encountered.

See also ATC, ATD, SS

ATT — A-Trap Trace

Syntax ATT[A] [*trap* [*trap*]] [*n* | *expr*]

Description The ATT command displays information about the execution of the specified A-trap(s). Traps can be specified by either trap number or trap name. Appending the letter *A* to the ATT command tells MacsBug to display information only when the given trap is called from the application heap. (Note that this means the current application heap at the time the ATT command was entered.) Specifying two traps indicates a range of traps; MacsBug displays information for every trap encountered within this range. If no traps are specified, a default range of A000 through ABFF is used.

If *n* is specified, MacsBug displays information only after a given trap has been encountered *n* times. If *expr* is specified, MacsBug displays information only when a given trap has been encountered and *expr* is TRUE. If neither *n* nor *expr* is given, MacsBug displays information each time the trap is encountered.

Example

```
ATT HideCursor
 HideCursor      PC = 0000A6F8       D0 = 0057007D     A7 = 004A6EC0
```

See also ATC, ATD

BR — Breakpoint

Syntax

BR *addr* [*n* | *expr*] [';*cmds*']

Description

The BR command sets a breakpoint at the specified address. If *n* is specified, MacsBug breaks only after *addr* has been reached *n* times. If *expr* is specified, MacsBug breaks only when *addr* has been reached and *expr* is TRUE. If neither *n* nor *expr* is given, MacsBug breaks each time *addr* is reached. You can also supply one or more commands to be executed once the break conditions are satisfied; each command must be preceded by a semicolon.

Entering BR without any parameters displays the breakpoint table, a list of all breakpoints in the order in which they were set; see the description of the BRD command for details.

Warning: Setting a breakpoint at a ROM address will cause execution to be slow since MacsBug must trace through each instruction until the breakpoint address is reached.

Warning: You should be sure that the given address contains an instruction. MacsBug implements breakpoints by placing a `TRAP #F` instruction in the word at *addr*. If *addr* points to the middle of an instruction, the substituted `TRAP #F` instruction will be treated as part of the instruction, possibly causing an error.

Be aware that MacsBug stores the information for breakpoints, step commands, and A-trap commands in a single table. New entries are entered at the end of the table. It's possible to receive the error message "Entry will not fit in the table" while entering a BR command if step commands, A-trap commands, and other BR commands have already filled this table.

If you set a breakpoint in a relocatable block, MacsBug stores the breakpoint as a handle to the breakpoint address. This means that if the block moves, the breakpoint is updated automatically.

Examples

```
BR TestProc+10              {break when TestProc+10 is reached}
BR TestProc+20 3            {break when TestProc+20 is reached 3 times}
BR TestProc+30 D0 = 1       {break when TestProc+30 is reached and register D0=1}
BR TestProc+40 A0 <> 0 ';DM A0 40'   {break when TestProc+40 is reached and }
                                     { register A0 is not equal to 0; then display }
                                     { memory at address in A0 for 40 bytes}
```

For a display of the breakpoint table with these breakpoints set, see the BRD command.

See also BRC, BRD

BRC — Breakpoint Clear

Syntax

BRC [*addr*]

Description

The BRC command clears the specified breakpoint; if no parameters are specified, all breakpoints are cleared.

See also

BR, BRD

BRD — Breakpoint Display

Syntax BRD

Description The BRD command displays the breakpoint table, a list of all breakpoints in the order in which they were set.

If the BR command that set a breakpoint specified a break only after reaching the address *n* times, *n* is shown in the third column, preceded by the number of times the address has been reached so far. If an expression was entered instead, it's shown in the third column. The fourth column shows any commands that were specified for execution upon breaking into MacsBug.

Note: MacsBug implements the GT command by setting a temporary breakpoint. If you enter MacsBug by some other means and execute the BRD command, this breakpoint remains set and you'll see an entry for it in the breakpoint table.

In the example below, the following breakpoints were set with the BR command:

```
BR TestProc+10
BR TestProc+20 3
BR TestProc+30 D0 = 1
BR TestProc+40 A0 <> 0 ';DM A0 40'
```

Example

```
BRD
 Breakpoint table
  Address   Module name              Cur/Max or Expression  Commands
  004635E0  TestProc+10              00000000 / 00000001
  004635F0  TestProc+20              00000000 / 00000003
  00463600  TestProc+30             D0 = 1
  00463610  TestProc+40             A0 <> 0                 ;DM A0 40
```

See also BR, BRC

CS — Checksum

Syntax CS [*addr1* [*addr2*]]

Description The CS command computes a checksum for the memory range from *addr1* through *addr2* and saves the result. If *addr2* is omitted, it checksums the long word at *addr1*.

Subsequent CS commands without parameters recompute the checksum and compare it with the previous value. If no address range has been previously specified, entering CS without parameters will return the error message "Address range must be entered before comparisons."

DB — Display Byte

Syntax DB [*addr*]

Description The DB command displays the byte at the specified address. If *addr* is omitted, DB displays the byte at the dot address. Pressing Return displays the next byte. The dot address is always set to the address of the byte displayed.

Example

```
DB 0
{Return}

 Byte at 00000000 = $40      64      64    '@'
 Byte at 00000001 = $81     129    -127    '•'
```

See also DL, DM, DP, DW

DH — Disassemble Hexadecimal

Syntax DH *expr...*

Description The DH command disassembles the given expressions as a sequence of 16-bit opcodes. This command is useful in converting hexadecimal values to assembler mnemonics.

Example

```
DH 4E56 0000

 Disassembling hex value
              00308AB6    LINK        A6,#$0000                                     4E56 0000
```

DL — Display Long

Syntax DL [*addr*]

Description The DL command displays the long word at the specified address. If *addr* is omitted, DL displays the long word at the dot address. Pressing Return displays the next long word. The dot address is always set to the address of the long word displayed.

Example

```
DL 0
{Return}

 Long at 00000000 = $40810000     1082195968     1082195968     '@•••'
 Long at 00000004 = $40802A14     1082141204     1082141204     '@•*•'
```

See also DB, DM, DP, DW

DM — Display Memory

Syntax DM [*addr* [n | template|basic type]]

Description The DM command displays memory starting from the specified address and continuing for *n* bytes. If *n* is omitted, 16 bytes are displayed. If both *addr* and *n* are omitted, DM displays 16 bytes beginning at the dot address. Pressing Return displays the next 16 bytes. The dot address is always set to the address of the first byte displayed.

Instead of specifying a number of bytes, you can specify the name of a template or one of the basic types used in creating a template. See the TMP command for details.

Examples

```
DM 0

 Displaying memory from 0
  00000000   4081 0000 4080 2A14   004F 6306 4080 20FC   @•••@•"••Oc•3• •
```

Note that the "•" character represents nonprintable characters. In the next example, `windowList` is a macro defining a low memory global variable, and `WindowRecord` is a template.

```
DM windowList^ WindowRecord

 Displaying WindowRecord at 003A0B14
  003A0B24   portRect          0000 0000 01B3 027A
  003A0B2C   visRgn            003A3E88
  003A0B30   clipRgn           003A4570
  003A0B80   windowKind        0045
  003A0B82   visible           TRUE
  003A0B83   hilited           TRUE
  003A0B84   goAwayFlag        TRUE
  003A0B85   spareFlag         TRUE
  003A0B86   strucRgn          003A4584
  003A0B8A   contRgn           003A4598
  003A0B8E   updateRgn         003A45AC
  003A0B92   windowDefProc     20832A5C
  003A0B96   dataHandle        003A6154
  003A0B9A   titleHandle       HD:Examples
  003A0B9E   titleWidth        0052
  003A0BA0   controlList       003A4610
  003A0BA4   nextWindow        003A05E8
  003A0BA8   windowPic         NIL
  003A0BAC   refCon            003A07E0
```

See also DB, DL, DP, DW

DP — Display Page

Syntax DP [*addr*]

Description The DP command displays a page, or 128 bytes, of memory, starting from the specified address. If *addr* is omitted, DP displays bytes beginning at the dot address. Pressing Return displays the next 128 bytes. The dot address is always set to the address of the first byte displayed.

Example

```
DP 0

 Displaying memory from 0
  00000000  4081 0000 4080 2A14  004F 6306 4080 20FC  @•••@•*••Oc•@•  •
  00000010  4080 20FE 4080 2100  4080 2102 4080 2104  @•  •@•!•@•!•@•!•
  00000020  4080 2106 4080 2108  4080 64BA 4080 210C  @•!•@•!•@•d•@•!•
  00000030  4080 210E 4080 210E  4080 210E 4080 210E  @•!•@•!•@•!•@•!•
  00000040  4080 210E 4080 210E  4080 210E 4080 210E  @•!•@•!•@•!•@•!•
  00000050  4080 210E 4080 210E  4080 210E 4080 210E  @•!•@•!•@•!•@•!•
  00000060  4080 210E 0000 B010  4080 622E 4080 622E  @•!•••••@•b.@•b.
  00000070  4080 60D0 4080 612C  004D 0456 004D 0456  @•`•@•a,•M•V•M•V
```

See also DB, DL, DM, DW

DSC — Extended Discipline

Syntax

DSC [ON | OFF]

Description

The DSC command turns the Extended Discipline utility on and off; if no parameter is passed, the command acts as a toggle. This utility examines parameters before traps are called and checks results after the calls complete. If Extended Discipline detects an error, MacsBug is invoked. See the Extended Discipline manual for more details.

DV — Display Version

Syntax DV

Description The DV command displays the version number of MacsBug currently in use.

Example

```
DV
 MacsBug version 6.0
 Copyright Apple Computer, Inc. 1981-1988
```

DW — Display Word

Syntax DW [*addr*]

Description The DW command displays the word at the specified address. If *addr* is omitted, DW displays the word at the dot address. Pressing Return displays the next word. The dot address is always set to the address of the word displayed.

Example

```
DW 0
(Return)

 Word at 00000000 = $4081     16513     16513    '@•'
 Word at 00000002 = $0000         0         0    '••'
```

See also DB, DL, DM, DP

DX — Debugger Exchange

Syntax DX [ON | OFF]

Description By default, two traps, Debugger ($A9FF) and DebugStr ($ABFF), let you enter MacsBug from within your program. The DX command lets you turn these "user breaks" on and off; without parameters, it acts as a toggle.

Note: Even when user breaks are disabled, messages specified by DebugStr will still be displayed; commands associated with DebugStr, however, are ignored. Also, the DX command does not affect breakpoints, exceptions, or other A-traps.

EA — Exit to Application

Syntax	EA
Description	The EA command attempts to launch the current application again. The current application heap is freed and reallocated.
See also	ES

ES — Exit to Shell

Syntax ES

Description The ES command allows you to exit from the current application. It executes the ExitToShell trap, which launches the current shell (typically the Finder).

Note: The ES command may not work with applications that override system traps. ExitToShell initializes the application heap, usually destroying any system patches located there.

See also EA

F — Find

Syntax

F addr n expr | 'string'

Description

The F command searches the range *addr* to *addr+n-1* for the specified pattern. When passing a string, be aware that case is significant. If *expr* is given, the width of the pattern is the smallest unit (byte, word, or long word) that will contain the value. Pressing Return repeats the search for the next *n* bytes. The F command sets the dot address to the first byte of the pattern found.

In the example below, the string isn't found the first time. Pressing Return repeats the command and finds it. The dot address is set to 2E1.

Example

```
F 0 200 'Finder'
{Return}

 Searching for 'Finder' from 00000000 to 000001FF
  Not found
 Searching for 'Finder' from 00000200 to 000003FF
  000002E1  4669 6E64 6572 2020  2020 2020 2020 2000  Finder
```

G — Go

Syntax G [*addr*]

Description The G command is used to leave MacsBug and resume program execution. This command is most frequently used without an address to resume execution where you left off; in other words, at the current program counter. If *addr* is given, execution resumes at that address.

Command-G is provided as a shortcut. Note that any commands sitting in the command line are ignored.

See also GT, MR

GT — Go Till

Syntax GT *addr*

Description The GT command sets a breakpoint at *addr* and resumes execution until the program counter reaches that address.

Warning: Setting a breakpoint at a ROM address will cause execution to be slow since MacsBug must trace through each instruction until the breakpoint address is reached.

Note: MacsBug implements the GT command by setting a temporary breakpoint. If you enter MacsBug by some other means, this breakpoint remains set. (In fact, you can see an entry for it in the breakpoint table if you enter the BRD command.) Executing the G command will resume execution until the breakpoint is reached or another exception occurs.

MacsBug 6.0 comes with a predefined macro `'GTO'` that expands to `'GT :+'`. This macro is useful for executing code until an offset in the current procedure. For instance, typing `GTO 22` expands to `GT :+22`, with the colon expanding to the current procedure name.

See also G, MR

HC — Heap Check

Syntax

HC

Description

The HC command checks the consistency of the current heap and reports any errors. Heap integrity cannot be checked rigorously but is examined for certain telltale signs of corruption. The possible error conditions are given below.

Note that all the heap commands check the heap as they execute; if a heap error is detected, they cancel the operation and return the same error message that the HC command would return.

"Zone pointer is bad": The zone pointer for the current heap (SysZone, ApplZone, or user address) must be even and in RAM. In addition, the bkLim field of the header must be even and in RAM, and must point after the header.

"Free master pointer list is bad": Free master pointers in the heap are chained together, starting with the hFstFree field in the zone header and terminated by a NIL pointer.

"BlkLim does not agree with heap length": Walking through the heap block by block must terminate at the start of the trailer block, as defined by the blkLim field of the zone header.

"Block length is bad": The block header address plus the block length must be less than or equal to the trailer block address. Also, the trailer block must be a fixed length.

"Nonrelocatable block: Pointer to zone is bad": Block headers of nonrelocatable blocks must contain a pointer to the zone header.

"Relative handle is bad": The relative handle in the header of a relocatable block must point to a master pointer.

"Master pointer does not point at a block": The master pointer for a relocatable block must point at a block in the heap.

"Free bytes in heap do not match zone header": The zcbFree field in the zone header must match the total size of all the free blocks in the heap.

See also

ATHC, HD

HD — Heap Display

Syntax

HD [*qualifier*]

Description

The HD command displays information about blocks in the current heap. The following qualifiers can be specified:

F:	Free blocks
N:	Nonrelocatable blocks
R:	Relocatable blocks
L:	Locked blocks
P:	Purgeable blocks
RS:	Resource blocks
TYPE:	Resource blocks of this type only

If no qualifier is specified, information about all blocks is displayed. If you specify F, N, or R, MacsBug checks the tag byte of the block headers for blocks with the appropriate bit set. If you specify L, P, or RS, MacsBug checks the master pointers for blocks with the lock, purge, or resource bits set. (For more details, see the Memory Manager chapter of *Inside Macintosh.*)

You can also request information about resource blocks of a particular resource type only (for instance, 'CODE', 'CRSR', and so on) simply by specifying the type. It's not necessary to quote the resource type, unless you want MacsBug to distinguish between uppercase and lowercase characters.

If no blocks of a specified type are found, the HD command returns the message "No blocks of this type found."

An example of the heap display is provided below.

For each block, the first column (Start) of the display gives the start of the data in the block, and the second column (Length) gives the length of the block, not including the header. Blocks that cannot be moved (nonrelocatable or locked) are indicated by a "•" character before the start address.

The third column (Tag) indicates the status of the block as free (F), nonrelocatable (N), or relocatable (R). For relocatable blocks, the fourth column contains the master pointer, while the fifth and sixth columns indicate whether the block is locked (L) or purgeable (P).

For resource blocks, the resource type, resource ID, file reference number and resource name (if specified) are shown.

A summary of the heap is displayed when all heap blocks have been processed. The totals are the same regardless of whether or not a qualifier was specified. (Note that to display only the summary information, you can use the HT command.)

Example

```
HD

 Displaying the Application heap
      Start      Length     Tag  Mstr Ptr   Lock Purge  Type    ID    File   Name
   •0046321C   00000100    N
   •00463324   00000004    R    00463318    L
   •00463330   00000070    R    0046330C    L            CODE   0001   0294   Main
    004633A8   00000008    F
    004633B8   00000058    R     00463310
    00463418   00000078    R     00463314
    00463498   00000018    R    00463308          P      CODE   0000   0294
    004634B8   00000004    R     00463304
    004634C4   00001518    F

                                    Total blocks   Total size
  Free                                00000002      00001530
  Nonrelocatable                      00000001      00000108
  Relocatable                         00000006      00000188
    Locked                            00000002      0000007C
    Purgeable and not locked          00000001      00000020
  Heap size                           00000009      000017C0
```

See also HC

HELP — Help

Syntax HELP [*cmd* | *section*]

Description The HELP command displays information about the given command or section. If no parameter is passed, a list of section headings is displayed. Pressing Return displays each section in turn.

Note: The HELP information is contained in a resource of type 'mxbh' that's approximately 10K in size. If space is especially tight, you can remove this resource, thereby disabling the HELP command. Do not ever modify the resource, however, since the HELP command expects the information in a particular format.

Examples

```
HELP

 Return shows sections sequentially. "HELP name" shows that section.
   Editing
   Expressions
   Values
   Operators
   Flow control
   Breakpoints
   A-traps
   Disassembly
   Heaps
   Symbols
   Stack
   Memory
   Registers
   Macros
   Miscellaneous

HELP Stack

 Stack
  SC6 [addr]
      Show the calling chain based on A6 links. If no addr then the
      chain starts with with A6. If addr then the chain starts at addr.
  SC7
      Show possible return addresses on the stack. A return address is
      an even address that points after a JSR, BSR or A-trap.

HELP SD

  SD [↑][name]
      Display all symbols in the current heap that partially match name.
      If '↑' then display all trap names that partially match name. If
      no name then display all symbols or traps.
```

HS — Heap Scramble

Syntax

HS [*addr*]

Description

The HS command toggles heap scrambling on or off. When heap scrambling is on, all relocatable blocks in the heap will be moved (if possible) whenever the following traps are encountered: NewPtr, NewHandle, ReallocHandle, SetPtrSize, or SetHandleSize. With SetPtrSize and SetHandleSize, the heap is scrambled only if the block size is being increased.

The only blocks not moved are single blocks between two stationary blocks. The heap is checked before scrambling; if it has been corrupted, MacsBug breaks and reports the error. (See the HC command for a list of possible errors.) Heap scrambling is automatically turned off when a bad heap is detected.

You can specify the address of the heap to be scrambled; if you don't, the address contained in the global variable ApplZone (the beginning of the application heap) is used.

See also

HC

HT — Heap Totals

Syntax HT

Description For the current heap, the HT command displays the total number of each type of block, the heap size, and the free space in the heap.

Example

```
HT

 Displaying the Application heap
                          Total blocks        Total size
  Free                         00000002            000010B8
  Nonrelocatable               00000001            000001C8
  Relocatable                  00000007            00000600
    Locked                     00000003            000004D4
    Purgeable and not locked   00000001            00000028
  Heap size                    0000000A            000017C0
```

See also HD

HX — Heap Exchange

Syntax

HX [*addr*]

Description

The HX command sets the current heap for the other heap commands. The address of a heap zone can be specified by *addr*. If no parameter is specified, the HX command cycles between the application heap, the system heap, and any other heap specified by a previous HX command.

Note: The name (or address) of the current heap is shown in the status region of the MacsBug display.

See also

HC, HD, HT, HZ

HZ — Heap Zones

Syntax

HZ

Description

In a system running MultiFinder, there will be an application heap for each application. The HZ command displays the addresses of all known heap zones. It identifies heaps by doing a heap check on each block in the MultiFinder heap; if the block passes, it's assumed to be a heap. The HZ command will not display heap zones stored on the stack or in the system heap, nor will it find heap zones that don't start at the beginning of a heap block.

See also

HC, HD, HT, HX

ID — Disassemble One Line

Syntax

ID [*addr*]

Description

The ID command disassembles one line, starting at the specified address. If *addr* is omitted, the program counter is used. Pressing Return disassembles the next line. The dot address is set to the address specified.

See also

IL, IP, IR

IL — Disassemble From Address

Syntax IL [*addr* [*n*]]

Description The IL command disassembles *n* lines, starting at the specified address. If *addr* is omitted, the program counter is used. If *n* is omitted, half a screen of code is displayed. Pressing Return disassembles the next *n* lines (if *n* was specified initially) or the next half screen (if *n* was omitted). The dot address is always set to the address specified.

The procedure name and offsets are given in the first column, followed by the actual addresses. A "•" character after the address indicates that a breakpoint is set at that instruction. The next two fields contain the opcode and operand; a "*" character before the opcode indicates the current PC.

The comment field (;) gives the target of a JMP, JSR, or BSR instruction or the trap number of a trap. The last field shows the actual hexadecimal words of the instruction; if there are too many words, an ellipsis (...) is shown. Note that this last field is shown only on larger displays, but can be always be seen by sending the output to a file or printer with the LOG command.

Example

```
IL

 Disassembling from 00308A96
  Main
      +000C  00308A96   *JSR      PROCATLEVEL1+0000           ; 00308A6A   | 4EBA FFD2
      +0010  00308A9A    JSR      *+$0312                     ; 00308DAC   | 4EBA 0310
      +0014  00308A9E •  JSR      *+$0314                     ; 00308DB2   | 4EBA 0312
      +0018  00308AA2   RTS                                                | 4E75
      +001A  00308AA4   UNLK      A6                                       | 4E5E
      +001C  00308AA6   RTS                                                | 4E75
             00308AB4    DC.W     $0000                       ; '••'       | 0000
  _RTInit
      +0000  00308AB6   LINK      A6,#$0000                                | 4E56 0000
      +0004  00308ABA   MOVEM.L   D3/D6/D7/A3/A4,-(A7)                     | 48E7 1318
      +0008  00308ABE   MOVE.L    $0018(A6),D6                             | 202E 0018
      +000C  00308AC2   JSR       $002A(A5)                                | 4EAD 002A
      +0010  00308AC6   MOVEA.L   -$00AA(A5),A0                            | 206D FF56
      +0014  00308ACA   MOVE.L    $0008(A6),(A0)                           | 20AE 0008
      +0018  00308ACE   MOVEQ     #$01,D0                                  | 7001
      +001A  00308AD0   MOVEA.L   #$00000316,A0                            | 207C 0000
0316
      +0020  00308AD6   TST.L     (A0)                                     | 4A90
      +0022  00308AD8   BEQ.S     _RTInit+005A                ; 00308B10   | 6736
      +0024  00308ADA   MOVEA.L   #$00000316,A0                            | 207C 0000
0316
      +002A  00308AE0   MOVEQ     #$01,D1                                  | 7201
      +002C  00308AE2   AND.L     (A0),D1                                  | C290
```

See also ID, IP, IR

IP — Disassemble Around Address

Syntax IP [*addr*]

Description The IP command displays half a screen of disassembled code, centered around the instruction specified by *addr*. Pressing Return disassembles the next half screen. If *addr* is omitted, the program counter is used. The dot address is set to the first address displayed.

The procedure name and offsets are given in the first column, followed by the actual addresses. A "•" character after the address indicates that a breakpoint is set at that instruction. The next two fields contain the opcode and operand; a "*" character before the opcode indicates the current PC.

The comment field (;) gives the target of a JMP, JSR, or BSR instruction or the trap number of a trap. The last field shows the actual hexadecimal words of the instruction; if there are too many words, an ellipsis (...) is shown. Note that this last field is shown only on larger displays, but can be always be seen by sending the output to a file or printer with the LOG command.

Example

```
IP

 Disassembling from 00308A7C
  No procedure name
            00308A7C   ADDQ.W     #$2,A4                               | 544C
            00308A7E   DC.W       $4556                  ; ????        | 4556
            00308A80   DC.W       $454C                  ; ????        | 454C
            00308A82   MOVE.W     D0,-(A0)                             | 3100
            00308A84   DC.W       $0000,$4EBA                          | 0000 4EBA
            00308A88   DC.W       $02FE                  ; ????        | 02FE
 Main
    +0000   00308A8A   LINK       A6,#$0000                            | 4E56 0000
    +0004   00308A8E   MOVEA.L    (A7)+,A6                             | 2C5F
    +0006   00308A90   JSR        *+$02F8                ; 00308D88    | 4EBA 02F6
    +000A   00308A94   _Debugger                         ; A9FF        | A9FF
    +000C   00308A96  *JSR        PROCATLEVEL1+0000      ; 00308A6A    | 4EBA FFD2
    +0010   00308A9A   JSR        *+$0312                ; 00308DAC    | 4EBA 0310
    +0014   00308A9E • JSR        *+$0314                ; 00308DB2    | 4EBA 0312
    +0018   00308AA2   RTS                                             | 4E75
    +001A   00308AA4   UNLK       A6                                   | 4E5E
    +001C   00308AA6   RTS                                             | 4E75
            00308AB4   DC.W       $0000                  ; '..'        | 0000
 _RTInit
    +0000   00308AB6   LINK       A6,#$0000                            | 4E56 0000
    +0004   00308ABA   MOVEM.L    D3/D6/D7/A3/A4,-(A7)                 | 48E7 1318
```

See also ID, IL, IR

IR — Disassemble Until End of Procedure

Syntax IR [*addr*]

Description The IR command disassembles code beginning from the instruction specified by *addr*; if no address is given, the program counter is used. This command assumes that the specified instruction is part of a procedure. Code is disassembled until the end of the procedure. The dot address is set to the address specified.

The procedure name and offsets are given in the first column, followed by the actual addresses. A "•" character after the address indicates that a breakpoint is set at that instruction. The next two fields contain the opcode and operand; a "*" character before the opcode indicates the current PC.

The comment field (;) gives the target of a JMP, JSR, or BSR instruction or the trap number of a trap. The last field shows the actual hexadecimal words of the instruction; if there are too many words, an ellipsis (...) is shown. Note that this last field is shown only on larger displays, but can be always be seen by sending the output to a file or printer with the LOG command.

Example

```
IR :

 Disassembling from ':
  Main
     +0000  00308A8A   LINK      A6,#$0000                           | 4E56 0000
     +0004  00308A8E   MOVEA.L   (A7)+,A6                            | 2C5F
     +0006  00308A90   JSR       *+$02F8              ; 00308D88     | 4EBA 02F6
     +000A  00308A94   _Debugger                      ; A9FF         | A9FF
     +000C  00308A96  *JSR       PROCATLEVEL1+0000    ; 00308A6A     | 4EBA FFD2
     +0010  00308A9A   JSR       *+$0312              ; 00308DAC     | 4EBA 0310
     +0014  00308A9E • JSR       *+$0314              ; 00308DB2     | 4EBA 0312
     +0018  00308AA2   RTS                                           | 4E75
     +001A  00308AA4   UNLK      A6                                  | 4E5E
     +001C  00308AA6   RTS                                           | 4E75
```

See also ID, IL, IP

LOG — Log to a printer or file

Syntax

LOG [*pathname* | printer]

Description

The LOG command sends MacsBug output to a text file specified by *pathname* or to an ImageWriter printer via the serial port. MacsBug follows the hierarchical file system conventions; if you don't specify a pathname, it assumes the current directory. If the specified file doesn't already exist, it's created as an MPW text file, which can be opened from word processing applications as well as from MPW. If the specified file already exists and is of type "Text", LOG appends MacsBug output to what's already there. To turn logging off, simply type LOG without parameters.

Note: The LOG command does not work with the LaserWriter® driver, so you can't send MacsBug output directly to a LaserWriter. You can, of course, send the output to a file and then print it on a LaserWriter.

Warning: MacsBug, by design, uses as little of the system as possible; the LOG command violates this design criterion. Logging may not work depending on the state of the file system during your debugging session. Also, logging enables interrupts briefly while executing its low-level calls. If your program depends on interrupts being completely disabled, you should not use the LOG command.

Warning: If you log to a file while MPW is running, or while an application is running under MultiFinder, be aware that the log file will be closed when you leave MPW or quit the application.

MC — Macro

Syntax

MC *name*'*expr*' | *expr*

Description

The MC command creates a macro with the given *name* that expands to '*expr*' or to the current value of *expr*. If *expr* is not quoted, it is evaluated and converted to a string before being entered.

Macros are expanded before the command line is executed; thus they can contain anything you can type in a command line. You can use macros to create command name aliases, reference global variables, and name common expressions.

Warning: MacsBug expands all macros on the command line before interpreting any commands. You cannot define a macro and then reference it on the same line since the reference will be undefined at the time the macro is expanded.

MacsBug lets you define your own set of macros as resources of type 'mxbm' to be loaded at boot time. MacsBug reserves resource IDs 100 and 101 for its standard macros, which include macros for several hundred common global variables. The file Macros.r, included on the MacsBug disk, can be used as a model for building your own 'mxbm' resources.

Two macro names have been predefined by MacsBug for customizing the debugging environment. If there are certain commands you want executed the first time MacsBug is entered (such as SHOW, SWAP, LOG, SX, HX, and DX), define them as a macro called FirstTime in an 'mxbm' resource. (Remember that multiple commands must be separated by semicolons.) When MacsBug is first invoked, it loads the specified commands into the command line and executes them. This lets you configure MacsBug using your preferred settings. A second macro, called EveryTime, can be defined in a resource file or on the fly with the MC command. The commands specified by this macro will be executed each time, except the first time, MacsBug is invoked.

MacsBug treats commands defined by macros just like commands that you enter explicitly. If you create an EveryTime macro, be aware that the last command executed by that macro is set as the default command; this command will be repeated if you press Return.

Examples

```
MC Frame 'A6+10'
```

This example gets the current value of register A6 each time the Frame macro is expanded, and adds 10 to it.

```
MC Save CurrentA5
```

This example remembers the current value of this global variable. You could change it and then restore it by typing:

```
SL CurrentA5 Save
```

See also MCC, MCD

MCC — Macro Clear

Syntax MCC [*name*]

Description The MCC command clears the macro with the given name. If no name is specified, all macros are cleared.

See also MC, MCD

MCD — Macro Display

Syntax MCD [*name*]

Description The MCD command lists those macros that match the given name. If no name is specified, all macros are listed, including both predefined macros loaded from resource files and macros defined during the current debugging session. MacsBug provides partial name matching, returning all macros that begin with the specified name. If you enter `MCD Cur`, for instance, all names that start with "Cur" are shown.

Example

```
MCD Cur
 Macro Table
  Name                  Expansion
  CurActivate          A64
  CurApName            910
  CurApRefNum          900
  CurDeactive          A68
  CurDirStore          398
  CurJTOffset          934
  CurMap               A5A
  CurPageOption        936
  CurPitch             280
  CurrentA5            904
  CurStackBase         908
```

See also MC, MCC

MR — Magic Return

Syntax

MR [*offset* | *addr*]

Description

If you've stepped into a procedure and want to get out, you can use the MR command. It sets a temporary breakpoint at the first instruction after the call to the current procedure by replacing the return address on the stack with a MacsBug address. When the procedure returns, MacsBug gets control. It then performs an RTS in trace mode, breaking at the instruction after the call.

If no parameter is specified, the return address is assumed to be on the top of the stack. If specified, the parameter is interpreted relative either to register A7 or A6. If the parameter is less than the contents of A6, MacsBug assumes that it's an offset from register A7. If the parameter is equal to register A6, it's assumed to be a frame pointer for the current procedure. If the parameter is greater than register A6, it's interpreted as an offset for a procedure higher on the stack.

If the specified address is not in the range between A7 and CurStackBase, the error message "This address is not a stack address" is returned. Also, MacsBug checks that the specified address is in fact a valid return address, in other words, that it immediately follows a JSR, BSR, or A-trap instruction. If this is not the case, the error message "The address on the stack is not a return address" is returned.

Examples

If you are at the first instruction in a procedure, simply typing `MR` will break when the procedure is done.

If you are past the LINK A6 instruction, `MR A6` will break when the procedure is done. With nested procedures, `MR A6^` will break when the procedure that called the procedure you are in is done.

See also G, GT

RB — Reboot

Syntax RB

Description The RB command unmounts the boot volume and reboots the system.

See also EA, ES, RS

Registers

Syntax registerName [= | :=expr]

Description Entering a register name displays the register's value. Values can be assigned to registers by using either the "=" or the ":=" operator.

MacsBug uses the Motorola names for all registers; a list of these names is given below.

68000 Registers

Dn	Data Register n
An	Address Register n
PC	Program Counter
SR	Status Register
SP	Stack Pointer
SSP	Supervisor Stack Pointer

68020 Registers

ISP	Interrupt Stack Pointer
MSP	Master Stack Pointer
VBR	Vector Base Register
SFC	Source Function Code Register
DFC	Destination Function Code Register
CACR	Cache Control Register
CAAR	Cache Address Register

68030/68851 Registers

CRP	CPU Root Pointer
SRP	Supervisor Root Pointer
TC	Translation Control Register
PSR	PMMU Status Register

68881 Registers

FPn	Floating-Point Data Register n
FPCR	Floating-Point Control Register
FPSR	Floating-Point Status Register
FPIAR	Floating-Point Instruction Address Register

RN — Set Reference Number

Syntax

RN [*expr*]

Description

The RN command lets you restrict symbol references to the file whose reference number is specified by *expr*. The reference number can be found with either the HD or SD commands. If no expression is specified, the reference number of the current resource file, contained in the global variable CurMap, is used.

The RN command is useful when you're dealing with multiple files with the same symbol names. When you're working with MPW tools, for instance, there may be multiple code segments with the same name. Once you've specified a reference number with the RN command, subsequent symbol references are restricted to the file with a matching reference number.

Specifying 0 for *expr* restores the default situation where all symbols match.

See also

SD, SX

RS — Restart

Syntax RS

Description The RS command restarts the system as if the Restart menu item had been selected from the Finder.

See also EA, ES, RB

S — Step

Syntax

S [*n* | *expr*]

Description

The S command steps through the next *n* instructions or until the specified expression is TRUE. If neither parameter is specified, the S command simply steps through the next instruction. In contrast to the SO command, the S command will actually trace into the ROM when a trap is encountered.

An S command entered with a specified range or number of instructions (for instance `s 10`) might encounter a breakpoint while executing. If this happens, the break into MacsBug terminates the S command.

Command-S is provided as a shortcut. Note that any commands sitting in the command line are ignored.

Warning: Stepping through certain MMU instructions can cause MacsBug to hang. If you're doing MMU programming, be aware that MacsBug executes many instruction while executing an S command and expects a valid memory mapping.

See also

SO

SB — Set Byte

Syntax SB *addr* (*expr* | '*str*...)

Description The SB command assigns values to bytes, starting at *addr*. Expressions are evaluated to 32-bit values, and the low-order byte is used. Strings of any length (limited only by the length of the command line) can also be specified; the characters are placed in successive bytes. The dot address is set to the address of the first byte set.

In addition to setting the dot address, the SB command sets DM as the default command; pressing Return after having executed an SB command will display the memory just set.

Example

```
SB 0 1 222 33333
{Return}

 Memory set starting at 00000000
  00000000  0122 3300 0000 0000  0000 0000 0000 0000  •"3•••••••••••••
```

See also SL, SW

SC6 — Stack Crawl (A6)

Syntax SC6 [*addr*]

Description The SC6 command displays the stack frame and address of the current procedure and all procedures above it in the calling order.

The SC6 and SC7 commands must have a range of memory to constrain the search for frames or return addresses. They assume that register A7 is even and points to the top of the stack, and that the global variable CurStackBase points to the bottom of the stack. If any of these conditions is not met, the following error message is returned: "Damaged stack: A7 must be even and <= CurStackBase."

The SC6 command also assumes that register A6 or the parameter is the address of a frame on the stack and that it points within the range between register A7 and CurStackBase. If these conditions aren't met, the error message "A6 does not point to a stack frame" is returned.

Note: For historical reasons, SC is provided as an alias for the SC6 command.

Example

In this example, 4CEDE4 was the value of A6 at the time ProcAtLevel1 called ProcAtLevel2. 4CEDDC was the value of A6 at the time ProcAtLevel2 called ProcAtLevel3. The current value of A6 defines the stack frame for ProcAtLevel3.

```
SC6

 Calling chain using A6 links
  A6 Frame    Caller
   <main>     00041FAA MAINPROC+000C
  004CEDE4    00041F82 PROCATLEVEL1+0004
  004CEDDC    00041F66 PROCATLEVEL2+0004
```

See also SC7

SC7 — Stack Crawl (A7)

Syntax SC7

Description The SC7 command displays a possible calling chain with the stack addresses that contain each caller's return address. A return address must be even and a valid RAM or ROM address, and it must point immediately after a JSR, BSR, or A-trap instruction.

The SC7 command will almost certainly include old or invalid values (in other words, addresses not in the current calling chain), since local stack variables can change the stack top without changing the contents. You can use the frame and return addresses to examine the stack yourself; you can also use the addresses in other MacsBug commands.

The SC6 and SC7 commands must have a range of memory to constrain the search for frames or return addresses. They assume that register A7 is even and points to the top of the stack, and that the global variable CurStackBase points to the bottom of the stack. If any of these conditions is not met, the following error message is returned: "Damaged stack: A7 must be even and <= CurStackBase."

The first column shows possible return addresses. The second column shows the addresses of possible A6 frame values.

A frame address can be used as a parameter to SC6 to tell it where the A6 links start. For instance, typing `SC6 4CEDD4` will show the same calling chain as in the SC6 example. This is useful while debugging routines that don't use the standard A6 frame conventions.

SC7 shows a superset of the calling chain. SC6 can then be used to show the true calling chain at the point where SC7 finds the first valid frame.

Example

```
SC7

 Return addresses on the stack
  Stack Addr  Frame Addr   Caller
   004CEDEC                4080D5CC Chain+014E
   004CEDE8                00041FAA MAINPROC+000C
    004CEDE0    004CEDDC   00041F82 PROCATLEVEL1+0004
    004CEDD8    004CEDD4   00041F66 PROCATLEVEL2+0004
```

See also SC6

SD — Symbol Display

Syntax

SD [†] [*name*]

Description

The SD command displays a list of trap names or symbols in the current heap. MacsBug provides partial name matching, returning traps and symbols that begin with the specified name. If you enter `SD Sys`, for instance, all names that start with "Sys" are shown. Depending on the parameters supplied, the following information is displayed:

Command	Effect
SD name	Shows all symbols that partially match *name*
SD † name	Shows all traps that partially match *name*
SD	Shows all symbols
SD †	Shows all traps

Examples

```
SD †Sys

 Displaying trap names
  Trap   Address    Name
  A090   0000D2D0   SysEnvirons
  A9B2   004D027E   SystemEvent
  A9B3   004D0276   SystemClick
  A9B4   004D028E   SystemTask
  A9B5   408151BA   SystemMenu
  A9C2   004D026E   SysEdit
  A9C8   40805DCA   SysBeep
  A9C9   004D045E   SysError

SD Sys

 Displaying symbols from the Application heap
  Type    ID    File   Address    Proc name
  CODE   0003   0236   00394B08   SysMessage
  CODE   0004   0236   0039C6B8   SYSGROWZONE
  CODE   0004   0236   0039C748   sysLaunch
  CODE   0004   0236   0039CB56   sysRun
  CODE   0004   0236   0039CFAC   sysSignal
  CODE   0004   0236   0039E90C   sysTerm
  CODE   0006   0236   003C113A   sysSuspend
  CODE   0002   0236   00479BB6   sysInit
```

See also RN, SX

SHOW — Show

Syntax

SHOW *addr* | '*addr*' [L | W | A]

Description

By default, MacsBug displays the stack pointer at the top of the status region, as well as the bytes starting at that address. The address is evaluated each time the display is updated. The number of bytes displayed varies with the screen size and the format of the display. The SHOW command lets you specify the display in word, long word, and ASCII format, by passing W, L, or A respectively.

The SHOW command also lets you specify another area of memory for display. When *addr* is quoted, the specified address is evaluated each time the display is updated. If *addr* is not quoted, the address is evaluated once and the resulting address is always shown.

To restore the default display, enter `SHOW 'SP' L`.

Examples

```
SHOW 'A6+8'
```

This example shows the stack above the previous A6 value and return address; for routines using `LINK A6`, this will be the routine parameters.

```
SHOW curApName A
```

This example will always show the data at the address defined by the macro curApName.

SL — Set Long

Syntax SL *addr* (*expr* | '*str*...)

Description The SL command assigns values to long words, starting at *addr*. Expressions are evaluated to 32-bit values. Strings of any length (limited only by the length of the command line) can also be specified; the characters are placed in successive bytes. The dot address is set to the address of the first long word set.

In addition to setting the dot address, the SL command sets DM as the default command; pressing Return after having executed the SL command will display the memory just set.

Example

```
SL 0 1 222 33333
(Return)

 Memory set starting at 00000000
  00000000  0000 0001 0000 0222  0003 3333 0000 0000  •••••••"••33••••

SL 0 12 'Test'
(Return)

 Memory set starting at 00000000
  00000000  0000 0012 5465 7374  0000 0000 0000 0000  ••••Test••••••••
```

See also SB, SW

SM — Set Memory

Syntax SM *addr* (*expr* | '*str*' ...)

Description The SM command assigns values to memory starting at *addr*. The size of each assignment is determined by the value. Specific assignment sizes can be set by using the SB, SW, and SL commands.

In addition to setting the dot address, the SM command sets DM as the default command; pressing Return after having executed the SM command will display the memory just set.

Examples

```
SM 0 1 222 33333
(Return)

 Memory set starting at 00000000
  00000000  0102 2200 0333 3300  0000 0000 0000 0000  ••"••33•••••••••

SM 0 4 'Test'
(Return)

 Memory set starting at 00000000
  00000000  0454 6573 7400 0000  0000 0000 0000 0000  •Test•••••••••••
```

See also SB, SL, SW

SO — Step Over

Syntax

SO [*n* | *expr*]

Description

The SO command steps through the next *n* instructions or until the specified expression is TRUE. If neither parameter is specified, the SO command simply steps through the next instruction. In contrast to the S command, SO steps over traps, JSRs, and BSRs, treating them as a single instruction.

When stepping over a toolbox trap with the auto-pop bit set, MacsBug correctly returns to the address on the top of the stack at the time of the trap call (instead of to the address immediately after the trap).

An SO command entered with a specified range or number of instructions (for instance `so 10`) might encounter a breakpoint or some other exception while executing. If this happens, the break into MacsBug terminates the SO command. The SO command cannot be terminated if a trap, JSR, or BSR is being stepped over. In this case, MacsBug displays a warning and prevents the user from entering another step command until this one is completed.

Note: For historical reasons, T (for Trace) is provided as an alias for the SO command. In addition, Command-T is provided as a shortcut; note that any commands sitting in the command line are ignored.

Warning: Stepping through certain MMU instructions can cause MacsBug to hang. If you're doing MMU programming, be aware that MacsBug executes many instructions while executing an SO command and expects a valid memory mapping.

See also

S

SS — Step Spy

Syntax SS *addr1*[*addr2*]

Description The SS command is a variation on the S command that lets you keep track of a particular area of memory. For the range between *addr1* and *addr2*, the SS command calculates a checksum before executing the next instruction. If the checksum value changes, MacsBug is invoked. If *addr2* is omitted, SS waits for the long word at *addr1* to change.

The SS command is terminated on the next entry into MacsBug.

The SS command is optimized for speed with a long word; with longer checksum ranges, it can be slow. Programmers needing to watch large ranges may want to use a hardware emulator.

You can also use the SS command as a way of slowing down certain routines, those that draw to the screen, for instance, so you can actually watch how they work.

Example

This example specifies a range that will not change and can be used to watch drawing to screen.

```
SS  ROMBase^(RomBase^+40)
```

See also CS

SW — Set Word

Syntax SW *addr* (*expr* | '*str*'...)

Description The SW command assigns values to words starting at *addr*. Expressions are evaluated to 32-bit values, and the low-order word is used. Strings of any length (limited only by the length of the command line) can also be specified; the characters are placed in successive bytes. The dot address is set to the address of the first word set.

In addition to setting the dot address, the SW command sets DM as the default command; pressing Return after having executed the SW command will display the memory just set.

Example

```
SW 0 1 222 33333
{Return}

 Memory set starting at 00000000
  00000000  0001 0222 3333 0000  0000 0000 0000 0000  •••"33••••••••••

SW 0 12 'Test'
{Return}

 Memory set starting at 00000000
  00000000  0012 5465 7374 0000  0000 0000 0000 0000  ••Test••••••••••
```

See also SB, SL

SWAP — Swap Frequency

Syntax

SWAP

Description

The SWAP command controls the frequency of display swapping between MacsBug and the application, depending on whether the system is configured for a single screen or for multiple screens.

For single screens, the SWAP command toggles between drawing step and A-trap trace information to the MacsBug display without swapping the screen, and drawing the information and swapping each time.

For multiple screens, the SWAP command toggles between having the MacsBug screen always visible, and having the MacsBug screen visible only at break.

With multiple screens, MacsBug uses the "Welcome to Macintosh" screen by default. You'll probably want your application on the larger screen and MacsBug on the smaller screen. To select a different screen for the MacsBug display, press the Option key while clicking on the Monitor icon from the Control Panel and then drag the Macintosh icon to the desired screen.

SX — Symbol Exchange

Syntax

SX [ON | OFF]

Description

The SX command toggles between allowing and not allowing symbol names in place of addresses. By default, symbol names can be used anywhere an address is used as a command line parameter. MacsBug translates this name into an address by searching the current heap for a matching procedure name. MacsBug also displays disassembled code as offsets relative to a procedure. Since this search process can be slow, MacsBug provides a way to disable it.

See also

IL, RN, SD

TD — Total Display

Syntax TD

Description The TD command displays all CPU registers in the command region. Since most 68000 registers are constantly displayed in the status region, this command is useful for remembering the register values between commands.

To display the 68030 MMU registers, use the TM command.

Examples

```
TD (on a Macintosh Plus)
 68000 Registers
  D0 = 00000000      A0 = E0025470      USP  = FFFFFFFF
  D1 = 00000006      A1 = 000CC7B2      SSP  = 000CC6CA
  D2 = FFFF0040      A2 = 000CC7B2
  D3 = 00000000      A3 = 000CC7B2
  D4 = 00000000      A4 = 000213B2
  D5 = 00000000      A5 = 000CD594
  D6 = 00000000      A6 = 000CC6E4      PC   = E002547E
  D7 = 00000000      A7 = 000CC6CA      SR   = Smxnzvc       Int = 0

TD (on a Macintosh II)
 68020 Registers
  D0 = 00000000      A0 = E0017EA8      USP  = D72B5FFA
  D1 = 00000006      A1 = 00487290      MSP  = 234B30CD
  D2 = FFFF280C      A2 = 00487290      ISP  = 004871C6
  D3 = 00000000      A3 = 00487290      VBR  = 00000000
  D4 = 0048FFFF      A4 = 004872D2      CACR = 00000001       SFC = 7
  D5 = 00000000      A5 = 004D013C      CAAR = 08281E55       DFC = 7
  D6 = 004D013C      A6 = 004871D6      PC   = E0017EB6
  D7 = 00000000      A7 = 004871C6      SR   = SmXnzvc        Int = 0
```

See also TF, TM

TF — Total Floating-Point

Syntax TF

Description The TF command displays all 68881 registers. (These registers are not shown in the status region.)

Example

```
TF (on a machine with a 68881)
 68881 Registers
  FP0   = 7FFF FFFFFFFF FFFFFFFF          NAN(255)
  FP1   = 7FFF FFFFFFFF FFFFFFFF          NAN(255)
  FP2   = 7FFF FFFFFFFF FFFFFFFF          NAN(255)
  FP3   = 7FFF FFFFFFFF FFFFFFFF          NAN(255)
  FP4   = 7FFF FFFFFFFF FFFFFFFF          NAN(255)
  FP5   = 7FFF FFFFFFFF FFFFFFFF          NAN(255)
  FP6   = 7FFF FFFFFFFF FFFFFFFF          NAN(255)
  FP7   = 7FFF FFFFFFFF FFFFFFFF          NAN(255)
          EE MC           CC QT ES AE
  FPCR  = 00 00     FPSR = 00 00 00 00     FPIAR = 00000000
```

See also TD, TM

TM — Total MMU

Syntax TM

Description The TM command displays the MMU registers common to the 68851 and 68030. (These registers are not shown in the status region.)

Example

```
TM (on a machine with a 68851)
 MMU Registers
  CRP = 7FFF020240800050        TC   = 80F84500
  SRP = 7F55D27300000100        PSR  = 2216
```

See also TD, TF

TMP — Templates

Syntax

TMP [*name*]

Description

The TMP command lists every template whose name matches the specified name. If no name is specified, all loaded templates are displayed by name. MacsBug provides partial name matching, returning all templates that begin with the specified name. If you enter `TMP My`, for instance, all names that start with "My" are shown.

MacsBug lets you define your own templates as resources of type 'mxwt' to be loaded at boot time. MacsBug reserves resource ID 100 for its standard templates. There are two ways of creating your own 'mxwt' resources. You can use the file Templates.r, included on the MacsBug disk, as a model for building a resource with MPW (and the Rez tool). You can also use ResEdit; a file called ResEdit Templates contains ResEdit templates for creating 'mxwt' resources.

Note: The 'mxbt' resource type is also supported. It's preferable to use the 'mxwt' type, however, since it can be created with either MPW or ResEdit and allows more template fields than the 'mxbt' type.

Templates are composed of fields. Each field consists of a name, a type, and a count. The basic types are:

Byte	Display in hexadecimal
Word	Display in hexadecimal
Long	Display in hexadecimal
SignedByte	Display in decimal
SignedWord	Display in decimal
SignedLong	Display in decimal
UnsignedByte	Display in decimal
UnsignedWord	Display in decimal
UnsignedLong	Display in decimal
Boolean	Display byte as TRUE (nonzero) or FALSE (0)
pString	Display a Pascal string
cString	Display a C string (zero-terminated)

For all of the basic types except pString, the count indicates the number of items of that type to display. For instance, a type of Word with a count of 4 can be used to display a Rectangle on one line. With pStrings, the count indicates the maximum string size and is used to compute the next field address. If the string is only as long as the actual number of characters, specify 0 for count and MacsBug will use the length byte to determine the end of the string.

The basic types listed above can also be used individually with the DM command. Several additional field types are used only in templates:

Text	Display a text string for count bytes. (Resource types, for instance, can be shown with the Text type and a count of 4.)
Skip	Skips over the next count bytes without displaying.
Align	Aligns to a word boundary (used after C or Pascal strings).
Handle	Dereferences and display in hex. This type is used to show the address of a data structure, rather than its contents.
^Type	Dereferences a pointer and displays using the specified basic type or template. The display is indented 2 spaces.
^^Type	Dereferences a handle and displays using the specified basic type or template. The display is indented 2 spaces.

If a template named Temp contains a field type of `^Temp` or `^^Temp`, MacsBug assumes the field is a link to another data structure of the same type. For instance, the WindowRecord template (provided in Templates.r) uses a field type of ^WindowRecord to dereference the pointer contained in the nextWindow field of the windowRecord. Pressing Return displays the next window in the window list.

Linked lists are zero-terminated. If a template contains than one field specifying a link, MacsBug uses the last field found.

WH — Where

Syntax

WH [*addr* | *trap*]

Description

The WH command returns information about the location of a given trap, symbol, or address. If no parameter is specified, the program counter is used. Given an address that's in ROM, WH looks for the trap nearest to and before that address, and returns the trap name as well as an offset from the start of the trap. If the address is in the system heap or application heap, WH returns the symbol (name and offset).

MacsBug will also attempt to map a given address to low memory global names. It does this by trying to convert macro values into numbers. If the value is a legal number and matches the given address, the macro name is returned.

If a trap name or number is specified, the WH command returns the trap name, the trap number, and the address of the trap. If a symbol name is specified, WH returns the address.

The WH command sets the dot address; thus `WH name` followed by `IL .` will disassemble the code at `name`.

In the example below, typing `WH` gets information about the PC. It is in the procedure MainProc at offset 000C. The heap block where this procedure was found is also shown. (See the HD command for details.)

Example

```
WH
 Address 000E7D36 is in the Application heap at MainProc+000C
 It is in this heap block:
      Start     Length    Tag   Mstr Ptr   Lock Purge   Type    ID    File
  •000E7CC8  000003D0    R    000E7CAC    L            CODE   0001   0236
```

Appendix A
Command Summary

Flow Control
G — Go
GT — Go Till
S — Step
SO — Step Over
SS — Step Spy
MR — Magic Return

Breakpoints
BR — Breakpoint
BRC — Breakpoint Clear
BRD — Breakpoint Display

A-Traps
ATB — A-Trap Break
ATT — A-Trap Trace
ATHC — A-Trap Heap Check
ATSS — A-Trap Step Spy
ATC — A-Trap Clear
ATD — A-Trap Display
ATR — A-Trap Record
ATP — A-Trap Playback
DSC — Extended Discipline

Disassembly Commands
IL — Disassemble From Address
IP — Disassemble Around Address
ID — Disassemble One Line
IR — Disassemble Until End of Procedure
DH — Disassemble Hexadecimal

Heap Commands
HX — Heap Exchange
HZ — Heap Zone
HD — Heap Display
HT — Heap Totals
HC — Heap Check
HS — Heap Scramble

Symbol Commands
RN — Resource Number
SD — Symbol Display
SX — Symbol Exchange

Stack Commands
SC6 — Stack Crawl (A6)
SC7 — Stack Crawl (A7)

Memory Commands
DM — Display Memory
TMP — Display all loaded templates
DP — Display Page
DB — Display Byte
DW — Display Word
DL — Display Long
SM — Set Memory
SB — Set Byte
SW — Set Word
SL — Set Long

Register Commands
TD — Total Display
TF — Total Floating-Point
TM — Total MMU

Macro Commands
MC — Macro Create
MCC — Macro Clear
MCD — Macro Display

Miscellaneous Commands
RB — Reboot
RS — Restart
ES — Exit to Shell
EA — Exit to Application
WH — Where
F — Find
CS — Checksum
LOG — LOG (output to file or printer)
SHOW — Show (memory in the sidebar)
DV — Display Version
DX — Debugger Exchange
HELP — Display list of MacsBug commands
SWAP — Swap (screen display)

Appendix B
Error Messages

This appendix lists most of the error messages MacsBug can return.

"Unable to access that address"
"Addresses must be even"
Any command that takes an address parameter can get one of these errors. The first is a 68000 bus error exception, and the second is an address error exception.

"Value expected"
Some commands will supply default parameters when no parameter is specified. This error can be returned by commands that require certain parameters.

"Unrecognized symbol"
Any command that takes a symbol as parameter can receive this error if a valid symbol name could not be found in the heap and the name is not a valid trap name.

"Divide by zero error"
This error is returned when an expression attempts to divide a number by zero.

"Count must be greater than zero"
Any command that takes a count (BR, ATB) requires it to be greater than 0.

"Entry will not fit in the table"
MacsBug stores information about breakpoints, step commands, and A-trap commands in a single table. Note that it's possible to receive this message while entering one type of action for the first time (a breakpoint for instance), since other types of actions may have already filled this table.

"Damaged stack: A7 must be even and <= CurStackBase"
The stack commands (SC6, SC7) must have a memory range to constrain the search for frames or return addresses. They assume that register A7 is even and points to the top of the stack, and that the global variable CurStackBase points to the bottom of the stack.

"A6 does not point to a stack frame"
The SC6 command assumes that register A6, or the parameter if specified, is the address of the first frame on the stack. It must point within the range specified by register A7 and CurStackBase.

"This address is not a stack address"
The MR command can optionally take a parameter specifying where on the stack the return address for the current procedure is located. This address must be even and within the range specified by register A7 and CurStackBase.

"The address on the stack is not a return address"
The MR command must know where the return address for the current procedure is located on the stack, since it replaces this address with an internal MacsBug address. MacsBug checks that the address it replaces is in fact a return address. A return address is defined as an address immediately following a JSR, BSR, or A-trap instruction. (All forms of JSR and BSR are recognized.)

"Floating-point not allowed in expressions"
"64-bit registers not allowed in expressions"
All expressions are evaluated as unsigned 32-bit values; floating-point registers and some MMU registers cannot be evaluated in this context.

"No blocks of this type found"
The HD command was instructed to display only blocks of a specific kind and none were found.

"Address range must be entered before comparisons"
The CS command remembers a range of memory to checksum; subsequent CS commands compute the checksum and compare it against the previous value. If no address range has been previously specified, entering CS without parameters will return this message.

"Low address must be less than or equal to high address"
The CS command requires an ordered address range.

"MMU not installed"
The TM command functions only if the system has a 68851 or 68030 installed. This error also occurs if you try to display or set an individual MMU register.

"68881 not installed"
The TF command functions only if the system has a 68881 installed. This error also occurs if you try to display or set an individual floating-point register.

"Macro expansion exceeds maximum command line length"
Macros are expanded in the command line buffer. This is a fixed-length buffer determined by the width of the command line on the current display.

"The template contains an unrecognized basic type"
The field of the template currently being displayed is not a valid basic type; see the description of the TMP command for a list of all possible types.

"Templates cannot expand more than 8 levels"
Template definitions can themselves contain template definitions, and so on. Expansion is limited to eight levels. Since it's unlikely that a structure would contain this many levels, this message may indicate a template definition that contains a recursive path.

"PC is not inside a procedure"
The ":" character can be used to represent the address of the start of the procedure displayed in the program counter window. If you enter ":" and no symbol information can be found for the program counter, this error message will be displayed.

"Zone pointer is bad"
The zone pointer for the current heap (SysZone, ApplZone, or user address) must be even and in RAM. In addition, the bkLim field of the header must be even and in RAM, and must point after the header.

"Free master pointer list is bad"
Free master pointers in the heap are chained together, starting with the hFstFree field in the zone header and terminated by a NIL pointer.

"BlkLim does not agree with heap length"
Walking through the heap block by block must terminate at the start of the trailer block, as defined by the blkLim field of the zone header.

"Block length is bad"
The block header address plus the block length must be less than or equal to the trailer block address. Also, the trailer block must be a fixed length.

"Nonrelocatable block: Pointer to zone is bad"
Block headers of nonrelocatable blocks must contain a pointer to the zone header.

"Relative handle is bad"
The relative handle in the header of a relocatable block must point to a master pointer.

"Master pointer does not point at a block"
The master pointer for a relocatable block must point at a block in the heap.

"Free bytes in heap do not match zone header"
The zcbFree field in the zone header must match the total size of all the free blocks in the heap.

"Syntax error"
This is a "catch all" error message; it's used in cases where the error is obvious given the context of the command. Possibilities include:

- An expression contains a value, an operator, but no second value.
- A nested expression does not have matching parentheses.
- An address qualifier other than `.B`, `.W`, or `.L` has been given.
- An illegal character is in the command line.
- The ATSS command does not include an address range.
- The format parameter for the SHOW command is other than L, W, or A.
- The F command does not have the correct number of parameters.
- The value being assigned to a floating-point register is illegal.
- A toggle command has been passed a parameter other than ON and OFF.
- The HD command qualifier is not valid.

Appendix C
MacsBug Internals

MacsBug uses as little of the system as possible. In addition, when MacsBug gets control, it effectively halts the processor by disabling interrupts. This appendix gives details on the MacsBug implementation.

Beginning with the 128K ROM, support for debuggers is provided. When a system error or 68000 exception occurs, the ROM code examines the global variable MacJmp to see if a debugger is installed. The high-order byte of MacJmp is used to contain the following information:

Bit	**Meaning**
7	Set if debugger is running.
6	Set if debugger can handle system errors.
5	Set if debugger is installed.
4	Set if debugger can support the Extended Discipline utility.

If a debugger is installed, the register set is saved in the global variable SEVars, and a call is made to the address in the low-order 3 bytes of MacJmp. When the debugger returns, the register set is restored and execution returns at the address in the program counter.

While active, MacsBug installs a bus error handler to catch any illegal memory references. MacsBug does not install an address error handler since it can check if addresses are even before accessing them.

MacsBug itself forces two kinds of exceptions. The first is used in setting breakpoints. MacsBug replaces the first word in an instruction with a TRAP #F instruction; when the program reaches this point, an exception is generated. The second is used in tracing instruction execution while single-stepping. MacsBug forces an exception by setting the Trace bit of the status register before executing an instruction.

MacsBug installs its own trace exception handler whenever:

- At least one ROM breakpoint is set.
- A breakpoint was set at the PC when execution resumed. The instruction must be executed before the breakpoint can be reinstalled.
- A step command is in progress.
- A step spy command is in progress.

The SO command steps over JSR and BSR instructions by executing the call with the Trace bit set, replacing the return address with an address inside MacsBug and then proceeding normally. Stepping over a trap call is done by copying the trap instruction into MacsBug and proceeding from that point.

MacsBug installs its own A-trap exception handler whenever:

- An A-trap command is active.
- The Extended Discipline utility is enabled.
- Heap scrambling is enabled.
- It steps into a trap call.

Since interrupts are turned off, MacsBug gets keys by polling for a keyboard interrupt and then calling the interrupt routine at Lvl1DT+8. MacsBug fields the event by temporarily installing its own PostEvent handler.

MacsBug assumes the display on a Macintosh Plus or SE is at address $3FA700, accomodating external monitors that change ScrnBase. MacsBug always appears on the internal display.

On a Macintosh II, MacsBug uses the first item in the gDevList as its display. The device must support 1-bit mode, and the display is limited to 640 by 480 to conserve memory.

While swapping the user and MacsBug displays on multi-bit displays, MacsBug calls SetMode and SetEntries (using the Control trap) to set a bit depth of 1, and a black-and-white color table.

Appendix D
Debugger and DebugStr

This appendix shows how to declare and use the Debugger and DebugStr macros on a per language basis.

Assembly language

Declaration

```
_Debugger    OPWORD    $A9FF          ; predefined in the file ToolTraps.a
_DebugStr    OPWORD    $ABFF          ; not predefined - define yourself
```

Example calls

```
_Debugger                             ; enters MacsBug and displays user break message

STRING PASCAL                         ; Asm directive to make sure to push a
                                      ;  Pascal string
PEA #'Entered main loop'              ; push address of string on stack

_DebugStr                             ; enters MacsBug and displays message
```

Pascal

Declaration

```
{Defined in OSIntf.p (MPW version 2.0) or Types.p (MPW 3.0)}

PROCEDURE Debugger; INLINE $A9FF;
PROCEDURE DebugStr(str: str255); INLINE $ABFF;
```

Example calls

```
Debugger;                        {enters MacsBug and displays user break message}

DebugStr('Entered main loop');   {Enters MacsBug and displays message}
```

MPW C

Declaration

```
/*Defined in Strings.h (MPW version 2.0) or Types.h (MPW 3.0)*/
```

```
#include <strings.h>            /* Required for c2pstr() */

pascal void Debugger() extern 0xA9FF;
pascal void DebugStr(aString) char *aString; extern 0xABFF;
```

Example calls

```
Debugger();                     /*enters MacsBug and displays user break message*/

DebugStr("\pEntered main loop");
                                /*enters MacsBug and displays message*/
```

Appendix E
MacsBug 6.0 Highlights

For those already familiar with MacsBug, this appendix summarizes the differences between MacsBug version 5.5 and MacsBug 6.0.

Flow control

In MacsBug 5.5, the T command stepped through code, stepping over A-traps. MacsBug 6.0 steps over BSR and JSR instructions as well. The T command has been renamed SO (Step Over) to more clearly indicate its function; T is, however, still supported as an alias. In addition, Command-T is provided as a shortcut.

The ST command, which traced through instructions until reaching a specified address, has been removed. The GT command now performs this function. When passed an address in RAM, a breakpoint is set; when passed an address in ROM, GT traces through each instruction until the given address is reached.

Breakpoints

The BRD command displays the breakpoint table. Although using the BR command without parameters will still display the table, the BRD command was added to distinguish between the two operations of setting and displaying breakpoints.

The CL command, while still supported as an alias, has been replaced by the BRC command for mnemonic consistency with the BRD command.

A-traps

The A-trap commands have undergone the most modification in version 6.0. Several commands have been added, the existing commands have all been renamed for consistency, and the parameters and operation of certain commands have changed.

The names all begin with *AT* (for *A-trap*), followed by additional letters indicating the action, where *B = Break*, *T = Trace*, *R = Record*, *P = Playback*, *HC = Heap Check*, and *SS = Step Spy*. Appending the letter *A* to the ATB, ATHC, ATT, ATSS, and ATR commands restricts the command's action to traps in the application heap. Note that this means the current application heap at the time the trap command was entered.

The old A-trap commands and the new commands that replace them are as follows:

Old command	New command
AB	ATB
BA	ATBA
AT	ATT
AA	ATTA
AH	ATHC
*	ATHCA
AS	ATSS
*	ATSSA
AR	ATR
*	ATRA
AX	ATC
*	ATD
*	ATP
*	DSC

The parameters for the ATB, ATHC, ATT, and ATSS commands have changed. The trap ranges remain the same, except that MacsBug 6.0 adds support for multiple discontinuous ranges. The address range parameters have been replaced by a conditional expression. For instance, you can now enter `ATB HFSDispatch D0.W=6` to specify a break at the File Manager trap HFSDispatch only when the low-order word of register D0 (which contains the routine selector) has the value 6 (meaning DirCreate). Conditional breaks allow more flexibility, while still allowing the specification of an address range (for instance, `(addr1<=PC) AND (PC<=addr2)`. In the same way, the D0 parameters have been removed, since the value of a register can be checked with an expression as well.

The old AX command cleared all A-trap actions; the new ATC command lets you clear one or all A-trap actions.

The DSC command lets you turn the Extended Discipline utility on and off; this utility (soon to be available from APDA) examines the parameters passed to, and the results of, traps executed.

Disassembly

Two new commands let you selectively disassemble code. The IP command disassembles half a page of code, centered around a specified address. The IR command is provided for disassembling code until the end of a routine. Given an address, the IR command disassembles until it reaches the end of the module.

When disassembling code, MacsBug 6.0 displays negative register offsets as signed numbers. For instance, $FFF8(A5) is now –$0008(A5). Thus, to display this word with the DM command, you would now type `DM A5-8` instead of `DM <FFF8+A5`.

Note: With this new display, the primary use for the "<" character as a sign extension operator is gone. Since this usage of the symbol also conflicted with its usage as a "less than" operator in expressions, it is no longer accepted for sign extension.

Disassembled trap names now decode the Sys, Immed, and AutoPop bits.

Heaps

The HS command has been improved. In MacsBug 5.5, this command called the CompactMem routine to scramble the heap. MacsBug 6.0 uses a custom scrambling routine. The only blocks not moved by this command are single, relocatable blocks between two stationary blocks.

The HZ command displays all heap zones; this command is useful when you're running under MultiFinder. The HZ command will not find heaps created on stacks or in the system heap.

Symbols

The RN command lets you specify a file reference number in order to limit the search for symbols to a particular resource file. This command is especially useful when debugging MPW tools.

Stack

The SC command has been renamed SC6 to differentiate it from the new SC7 command. SC is still supported as an alias for SC6.

The SC7 command shows all *possible* return addresses on the stack, with their frames.

Memory

The DW, DL, and DP commands display respectively a word, long word, and page (128 bytes) of memory. The SW and SL commands let you set words and long words in memory.

The TMP command displays the names of all templates currently loaded. See the description of the TMP command in Chapter 3 for details on creating templates.

Registers

In MacsBug 5.5, the address and data registers were referred to as RAn and RDn, and the floating-point control registers were specified by FI, FC, and FS. MacsBug 6.0 uses the Motorola names for all registers. For instance, data register 3 is now specified as D3 (instead of RD3) and the floating-point control register is specified as FPCR (instead of FC). A list of Motorola register names is given in Chapter 3 under "Registers."

Note: The use of An and Dn for registers presents a conflict when you want to specify the hexadecimal numbers A0 through A7 and D0 through D7; in such cases, express the number explicitly as $An or $Dn. (Since registers are used much more frequently than these 16 hexadecimal numbers, the designers opted for this tradeoff.)

The contents of the CPU registers are always displayed in the status region of the MacsBug 6.0 display. The TD command, which displays the CPU registers in the command region, remains useful for comparing register contents between commands. The 68881 registers, not shown in the status region, can still be displayed with the TF command. The new TM command displays the contents of the MMU registers common to the 68851 and 68030.

Macros

Macros are a new feature of MacsBug 6.0. Macros are simple text string substitutions. They're expanded before the command line is executed, and can thus contain anything you can type in a command line. You can use macros to create command name aliases, reference global variables, and name common expressions.

Macros can be created on the fly with the MC command or they can be included in a resource of type 'mxbm' that's loaded at boot time. MacsBug reserves resource IDs 100 and 101 for its standard macros, which include macros for several hundred common global variables. The file Macros.r, included on the MacsBug disk, can be used as a model for building your own 'mxbm' resources.

Two macro names have been predefined by MacsBug for customizing the debugging environment. If there are certain commands you want executed the first time MacsBug is entered (such as SHOW, SWAP, LOG, SX, HX, and DX), define them as a macro called FirstTime in an 'mxbm' resource. (Remember that multiple commands must be separated by semicolons.) When MacsBug is first invoked, it loads the specified commands into the command line and executes them. This lets you configure MacsBug using your preferred settings. A second macro, called EveryTime, can be defined in a resource file or on the fly with the MC command. The commands specified by this macro will be executed each time MacsBug is invoked.

The MCD command displays a table listing all macros defined, including both those loaded at boot time and those defined with the MC command. The MCC command clears one or all macros.

Miscellaneous

Warning: In the past, a popular way to generate an exception was to add a line such as

```
DC.W  $FECE ; generate a line 1111 exception
```

at the point in your program where you wanted MacsBug to get control. (Any value $F000 through $FFFF could have been used.) This method should no longer be used, as these instructions have been reserved by Motorola for use in their coprocessor interface for the 68020 microprocessor.

Command-G, S, and T are provided as shortcuts for executing the G, S, and SO commands respectively.

The F command now takes different parameters. The search range is specified by a starting address and the number of bytes to be searched. (An ending address can no longer be used.) In addition, a mask is no longer specified. The F command can be repeated for the next *n* bytes by simply pressing Return.

The CV command, used to evaluate expressions, has been removed. MacsBug 6.0 evaluates anything on a command line that does not start with a command name.

The HELP command provides information on MacsBug commands. You can specify a single command or a group of commands, or simply browse through the entire HELP file.

The LOG command lets you send MacsBug output to a file or to an ImageWriter printer via the serial port.

Using the RS command has the same effect as selecting the Restart item from the Finder.

The SHOW command displays a specified area of memory in the status region of the MacsBug display.

The SWAP command lets you control the frequency of display swapping between MacsBug and the current application, depending on whether the system is configured for a single screen or for multiple screens.

MacsBug 6.0 adds the symbol "^" as a postfix indirection operator.

The TP command has been replaced by a macro named thePort; this macro is defined as `DM A5^^ WindowRecord`.

As an overview, a table showing the old commands that have been replaced, as well as the new commands, is given below.

Old command	New command or functionality
AA	ATTA
AB	ATB
AH	ATHC
AR	ATR
AS	ATSS
AT	ATT
AX	ATC
BA	ATBA
CL	BRC
CV	command line expression evaluation
SC	SC6
T	SO
ST	GT
TP	thePort macro
?	HELP

ATD	Display A-trap table
ATHCA	A-trap heap check (application heap)
ATP	Playback A-traps
ATRA	A-trap record (application heap)
ATSSA	A-trap step spy (application heap)
BRD	Display breakpoint table
DL	Display long word
DP	Display 128 bytes
DW	Display word
DSC	Toggle Discipline utility
IP	Disassemble half-page centered around address
IR	Disassemble until routine ends
HZ	List heaps
LOG	Send output to file or ImageWriter
MC	Define macro
MCC	Clear macro(s)
MCD	Display macros
RN	Resource file number filter for symbols
RS	Restart
SC7	Give procedure's possible calling chain and return addresses
SHOW	Display address in sidebar
SWAP	Toggle screen display
SL	Set long word
SW	Set word
TM	Display 68851 or 68030 MMU registers
TMP	List templates

Appendix F
Did You Know...?

This appendix contains tips, shortcuts, and interesting facts about MacsBug. Did you know that...

- Holding down the Control key forces a break into MacsBug immediately after it's loaded. This feature works only on Macintosh computers equipped with the Apple Desktop Bus™ (ADB) interface; the Control key was chosen since it's found only on ADB machines. On machines without ADB, the keyboard is loaded after MacsBug, so it makes no sense to break into MacsBug.

- The DebugStr routine with an argument of `';HC;G'` is a useful way to determine where in your program the heap may become corrupted. The HC command performs a heap check; if the heap is corrupted, MacsBug stops and reports the error. If the heap is in order, the G command is executed and program execution resumes. Sprinkling such calls to DebugStr throughout your program lets you hone in on memory culprits.

- A related technique is to use the ATHC command, which checks the heap prior to each trap call. Using this technique means that you don't need to modify your program, but it does have the disadvantage that you can't choose the frequency and location of the checks.

- In the same way that passing `';HC;G'` with DebugStr checks the heap, passing `';CS;G'` checksums a block of memory. If the block has changed, MacsBug takes over; otherwise program execution continues. Remember that the range must be set up with an initial CS command before subsequent CS commands can compare the checksum.

- You can create a custom A-trap trace by executing the ATB command with an associated action. For instance, you can specify the commands `';TD;G'` for execution upon break. Whereas the ATT command shows only select registers, this action displays all registers. You could further customize the trace by displaying memory based on the content of particular registers.

Appendix G
Procedure Definition

Whenever possible, MacsBug accepts and returns address as procedure names and offsets. Names are found by scanning relocatable heap blocks for valid procedure definitions. A procedure definition in the simplest case consists of a return instruction followed by the procedure's name.

A procedure is defined as:

> [LINK A6]
> Procedure code
> RTS or JMP(A0) or RTD
> procedure name
> procedure constants

The LINK A6 instruction is optional; if missing, the start of the procedure is assumed to be immediately after the preceding procedure, or at the start of the heap block.

The procedure name can be a fixed length of 8 or 16 bytes, or of variable length. Valid characters for procedure names are "a"–"z", "A"–"Z", "0"–"9", "_", "%", ".", and space. The space character is allowed only to pad fixed-length names to the maximum length.

With fixed-length format, the first byte is in the range $20 through $7F. The high-order bit may or may not be set. The high-order bit of the second byte is set for 16-character names, clear for 8-character names. Fixed-length 16-character names are used in object Pascal to show class.method names instead of procedure names. The method name is contained in the first 8 bytes and the class name is in the second 8 bytes. MacsBug swaps the order and inserts the period before displaying the name.

With variable-length format, the first byte is in the range $80 to $9F. Stripping the high-order bit produces a length in the range $00 through $1F. If the length is 0, the next byte contains the actual length, in the range $01 through $FF. Data after the name starts on a word boundary. Compilers can place a procedure's constant data immediately after the procedure in memory. The first word after the name specifies how many bytes of constant data are present. If there are no constants, a length of 0 must be given.

Examples of valid assembly-language procedure definitions are given below.

```
; Variable length name with no constant data.

Proc1        PROC
             LINK    A6, #0
             UNLK    A6
             RTS
             DC.B    $8C, 'VariableName'
             DC.W    $0000
             ENDP

; Fixed 8-character name.

Proc2        PROC
             LINK    A6, #0
             UNLK    A6
             RTS
             DC.B    $80 + 'F', 'ixed    '
             ENDP
```

```
; Fixed 16-character name.

Proc3          PROC
               LINK   A6, #0
               UNLK   A6
               RTS
               DC.B   $80 + 'M',  $80 + 'e', 'thod  Class    '
               ENDP
```

Index