MPW 3.2

Release Notes

Erratum

Due to an oversight, an erroneous comment symbol was used in lines 20 through 24 of the source file:

{MPW}PExamples:TESampleGlue.a

The character "#" was incorrectly written where the character ";" should have been written.

MPW 3.2
Release Notes

This release note summarizes the differences between MPW 3.2 and MPW 3.0, the last version for which a reference manual exists. Details beyond those given in this note are to be found in separate release notes entitled MPW 3.2 Shell, MPW 3.2 Object Pascal, MPW 3.2 C, MPW 3.2 Assembler, MPW 3.2 Libraries & Interfaces, MPW 3.2 "411" Help, MPW 3.2 Run-Time Architecture, MPW 3.2 SIOW, and MPW 3.2 Tools & Scripts. Three of the above: "411" Help, Run-Time Architecture, and SIOW describe completely new features.

Principal Changes

Shell

- Projector has a number of enhancements.
- The Editor provides split windows.
- A marker browser has been provided.
- There are additional commands for the `faccess` function.
- The `Save` command, when scripted, saves the resource fork of a file if any resource has changed.
- MPW can now be run under A/UX.
- New System 7.0 error codes have been added to the file SysErrs.err.

C

- A variety of new object code optimizations have been implemented.
- There are new compiler options to control optimization level.

- Support is provided for the MacApp debugger and code profilers (generating a preamble and postamble for functions).
- A new pragma is provided for passing parameters to functions via registers.
- A new pragma is provided to force the generation of MC68020 code.
- A new pragma is provided to prevent the multiple inclusion of header files.
- There is a new option for invoking the "32-bit everything" run-time architecture.
- A new pragma is provided to prevent dead code stripping by the linker.
- An option is provided for machines having the MC68020 and up that allows stand-alone code segments greater than 32K.

Object Pascal

- Further object code optimizations, including an optional two-pass code generation phase which allocates unused scratch registers for local data.
- An option to control optimization level.
- An option for invoking the "32-bit everything" run-time architecture.
- A new syntax for declaring forward and external objects
- Enhancements to the Object Pascal declaration handling to support MacApp.
- Support for USES clauses in the implementation section of a unit.
- Support for external C functions with arbitrary numbers of arguments,.
- The ability to omit static links for nested procedures that don't need them.
- Increase in the number of nested include files, the maximum number of nested compile-time conditionals, and in the number of long identifiers.

ASM

- New optimizations.
- Longer identifiers to support C++ name mangling.
- MC68040 support.
- A new parameter and syntax for module directives.
- "32-Bit Everything" support.

- Improved SADE support.
- A new macro function: &SYSINMOD.

Tools/Scripts

- The **Link**, **Lib**, **DumpCode**, and **DumpObj** tools support an improved form of data initialization.
- The **Search** tool has some new options and its performance has been improved.
- The **DumpCode** and **DumpObj** tools provide support for 68040 development.
- The **Object Pascal** compiler, **C** compiler, **ASM**, and **Link** support "32-bit Everything," which is a capability for simultaneously exceeding the 32K limit on code segment sizes, the size of the jump table, and the size of the application globals space.
- **Link** has an option which causes the automatic generation of "branch islands." This permits code segments to be larger than 32K.
- A new **StreamEdit** tool provides for scriptable editing of text streams similar to that provided by Unix's Sed tool.
- **Make** includes extensions to the default rule mechanism and new built-in variables.
- **Choose** has new options to prompt for "secure" passwords with a dialog box.
- **ProcNames** and **PasRef** now process "$" compiler directives such as conditional compilation directives. ProcNames has been modified to generate MPW Shell commands to mark all of the procedures and functions in a file.
- **FileDiv** has been enhanced to handle non-text files.
- **Compare** has been modified to support a larger number of lines in the files being compared.

Libraries/Interfaces

- CRuntime.o and Runtime.o have been merged into a single Runtime.o.
- The libraries have been resegmented to move more modules out of the "main" segment.
- The C libraries conform to the ANSI specification.
- The Pascal libraries include standard C string functions which work on Pascal strings.
- There is a new library supporting a simple input/output window application known as SIOW.

- The SANE libraries have performance enhancements.
- New Interfaces support several SANE enhancements and support System 7.0.
- CIncludes has been changed to eliminate glue code in calling certain Macintosh Toolbox routines.

Miscellaneous

- A new facility, **"411" Help**, provides rapid on-line access to development information while using MPW. Because of this size of the database, "411" Help will be distributed on CD only.

Important Note for Users of Virus Detectors

A number of virus detectors, e.g., Vaccine™ and SAM™ think that the MPW Linker is a virus because the Linker routinely creates and modifies 'CODE' resources. The behavior may range from the Linker appearing to hang to multiple appearances of a window requesting permission to proceed. This problem can possibly be corrected by changing the protection level of the virus detector. If all else fails, the detector can be temporarily disabled.

Important Note for Users of The Debugger V2 & MacNosy

Attempts to use the above products with MPW 3.2 will result in a bus error. The following change to the MPW Shell will remove the problem:

Using ResEdit, change the bytes at offsets $46B4 and $4722 in CODE resource ID 5 from $67 to $60.

Reporting Bugs

Please report any bugs you find to Apple. Use the application "Outside Bug Reporter," found on the MPW Installation Disk. After completing the bug report, send it

via AppleLink to: APPLE.BUGS

or

via US Mail to:

Apple Computer, Inc.
Bug Report Center
20525 Mariani Ave. MS 27-AN
Cupertino, CA 95014

- Apple Partners who are in critical need of a work-around solution should contact MacDTS

via AppleLink to: MACDTS

or

via US Mail to:

Apple Computer, Inc.
Developer Technical Support
20525 Mariani Ave. MS 75-3T
Cupertino, CA 95014

System Requirements

MPW 3.2 requires a hard disk and at least 2 Mb of RAM. Actually, the MPW Shell's MultiFinder partition size comes set at 2048K. In order to compile and link large programs with symbolic information, it may be necessary to increase MPW Shell's memory partition size. SADE requires MultiFinder and at least 2.5 Mb of RAM. MPW 3.2 requires System 6.0 and Finder 6.1 (or later).

MPW 3.2 Shell
Release Notes

MPW 3.2 is the first release since MPW 3.0 that goes significantly beyond bug fixes. The new MPW Shell incorporates safety features for Projector, split windows and miscellaneous improvements for the editor, an enhanced set of window commands, enhanced tool support via `faccess`, and a simple marker browser. A scripted `Save` command saves the resource fork of a file when any resource has changed. The "Mark" menu has been enhanced with the item "Alphabetic"; this toggles the list of marks between alphabetic order and the order of their appearance in the file The Shell is now truly 32-bit clean and runs also in the AUX 2.0 environment.

△ **Important** About testing tools: Memory management in the Shell has changed. The Shell no longer checks the heap before and after tools run; this is a consequence of following the rules for being "32-bit clean."

One way to see if tools are working correctly is to use MacsBug's HC command. For example, you can have this command executed every time a tool starts by using the following from inside MacsBug:

```
BR STARTTOOL '; HC; G'
```

This, however, has the disadvantage of causing the MacsBug window to flash momentarily on the screen. △

△ **Important** After GetFileName brings up the Standard File dialog when MPW is running in the background (under MultiFinder), switching MPW to the foreground will cause the computer to hang. The cause of this has not been determined, but it may be a System 7 problem. △

◆ *Note:* The default shell partition is now 2048K bytes (0x200000) and the default stack size is is 64K bytes (0x10000). Also, most work to be done under MPW 3.2 requires at least 256K more memory than it did under previous releases.

faccess Commands

The MPW Shell now contains code for several new `faccess` commands. The `#define` statements for the command names and the typedef for MarkElement are in {CIncludes}FCntl.h. The interface is:

```
int faccess(char *filename, unsigned int command, long *arg);
```

These commands require that the file be open.

The new commands are:

F_GSCROLLINFO — Sets `*arg` to the value 1 if scrolling is locked; to the value 0 if it is unlocked.

F_SSCROLLINFO — Locks scrolling if the value of `*arg` is non-zero; unlocks scrolling if it is zero.

F_SMARKER — Sets a file marker according to the specification in the structure `*arg`. This structure is defined by:

```
typedef struct MarkElement {
      int start; //start position of mark
      int end; //end position of mark*/
      unsigned char charCount; // no. of chars.
                               // in mark name
      char name[1]; // first char. of mark name
} MarkElement;
```

Note: The user must allocate sufficient contiguous space to hold an instance of MarkElement followed by a zero-terminated mark name. The necessary data items followed by the string must then be copied into this space.

F_SSAVEONCLOSE — Sets the status of "save on close." If `*arg` is 0 (SaveNormal), the normal prompt to the user is given. If `*arg` is 1 (SaveNever), the window is not saved. If `*arg` is 2 (SaveAlways), the window is saved.

F_GSAVEONCLOSE — Sets *arg to 0, 1, or 2, respectively, depending on whether the "save on close" status is SaveNormal, SaveNever, or SaveAlways.

(For calling instructions when using assembly language or Pascal, see Chapter 12 of the MPW 3.0 Reference Manual.)

Save

The `Save` command (scripted) will now force a save of the resource fork of a file if the format (font/tabs), scroll bar, window position, etc. has been modified, even if the text of the file has not been modified. Note that the "Save" menu item will continue to operate as before, including the fact that it will be disabled unless the text of the file has been modified.

Editor

Markers

A new item appears in the initial section of the "Mark" menu. It is a toggle labelled "Alphabetical." When a check mark appears next to this label, the marks shown in the pop-up menu will be in alphabetical order. Otherwise, the order is the sequence in which the marked lines appear in the window.

If the first character supplied for a marker name is a hyphen, the remaining characters are ignored, and the marker menu will show a menu separation line (a gray, one-pixel line). This line will appear in the menu exactly where a normal marker name, if supplied, would have appeared. The line will not show when the "Alphabetical" mode is selected from the Mark menu.

Keyboard Commands

The set of editing actions that are available from the keyboard has been extended. The set is shown in the matrix below.

The actions "move word to right/left" and "extend selection word to right/left" require precise definition. A word is defined to be a contiguous group of characters chosen from the set defined by the shell variable `WordSet`, the default value of which is `a-zA-Z_0-9`. The point to which the insertion point moves, or the selection is extended, is from the current position to the far edge of the nearest word in the direction of motion. The end of the line provides an exception. If there are characters not in `WordSet` after the last word found (in the direction of motion) on the line, then the cursor motion or selection extension stops at the furthest of these characters before proceeding to the next line (again, in the direction of motion). For example, given the following lines:

```
Now is the time for--
to come to the aid
```

consecutive applications of **option →** will move the cursor from the right-hand edge of "time" to the right-hand edge of "for" to just beyond "--" and then to the right-hand edge of "to".

	delete	del *	←	→	↑	↓
	deletes character to left	deletes character to right	moves character to left	moves character to right	move one line up	moves one line down
shift	deletes character to left	deletes character to right	extends selection character to left	extends selection character to right	extends selection one line up	extends selection one line down
option	deletes word to left	deletes word to right	moves word to left	moves word to right	moves one line up	moves one line down
shift option	beep	beep	extends selection word to left	extends selection word to right	extends selection one line up	extends selection one line down
cmd	deletes to end of file	deletes to end of file	moves to beginning of line	moves to end of line	moves one page up	moves one page down
cmd shift	beep	beep	extends selection to beginning of line	extends selection to end of line	extends selection one page up	extends selection one page down
cmd option	deletes to end of file	deletes to end of file	moves to beginning of line	moves to end of line	moves to beginning of file	moves to end of file
cmd shift option	beep	beep	extends selection to beginning of line	extends selection to end of line	extends selection to beginning of file	extends selection to end of file

* **del** is available on extended keyboard only

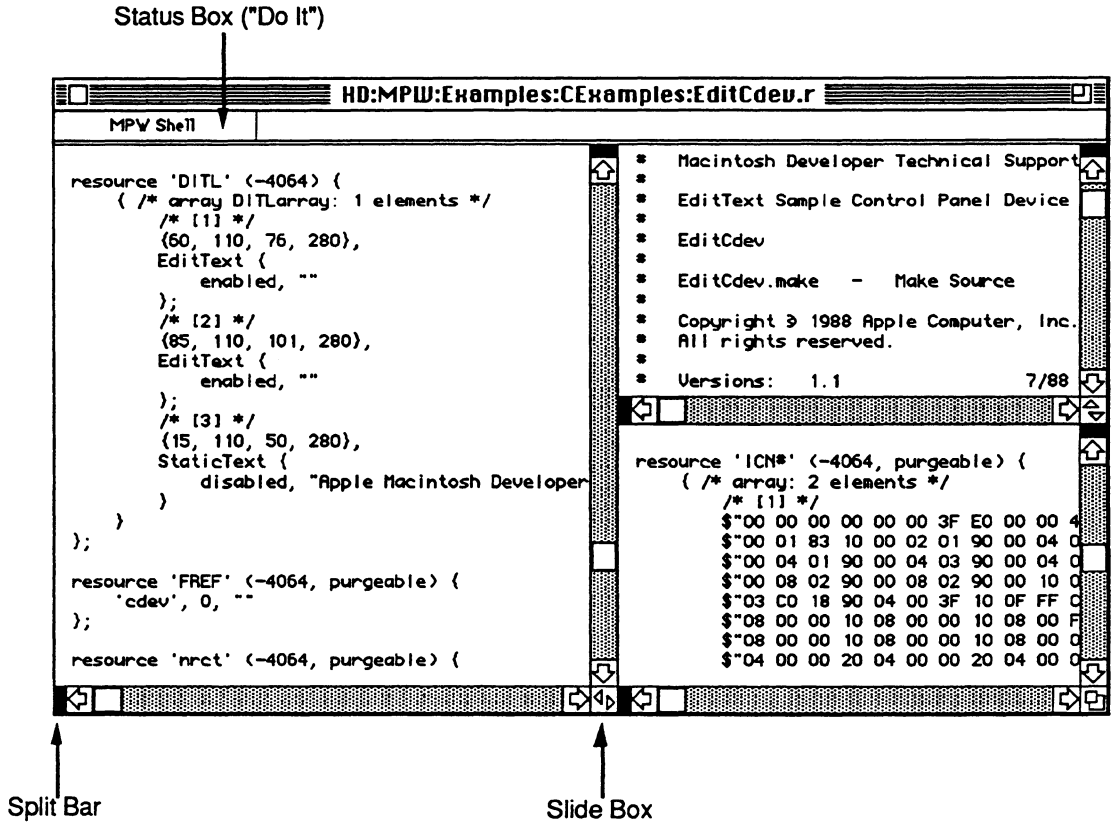
Windows

A new feature of the MPW Shell Editor is the capability to split windows. Each pane has both vertical and horizontal scroll bars. As specified in the *Human Interface Guide: The Apple Desktop Interface*, the user can create new panes by dragging the black rectangle (*split bar*) at the top/left of the scroll bar; the panes can be resized by dragging the *slide box* that is adjacent to the split bar or deleted by moving the slide box as far as possible in the direction of its origin. The split bar creates a new pane; the slide box resizes (possibly down to zero) an existing pane. Each pane is independently controllable (scroll and resize), which increases flexibility in viewing a text file.

Extensions have been made to the set of commands that deal with windows.

◆ *Note:* The "Do It" button has been moved to the top of the window.

- Editor Window



Window Commands

The **Windows** command has a new option, -o, that causes the pathnames to be written as a set of Open commands.

Syntax

```
Windows                 # list windows
Windows [-q] [-o]
      -q                # don't quote window names with
                        # special characters
      -o                # write out the "Open..." commands
```

The -o option causes the list to consist of a series of Open commands, one per line, containing the "-r" option for the case of read-only files.

◆ *Note:* The Suspend script has been modified to make use of `Windows -o`.

The **RotateWindows** command has a new option that reverses the direction of rotation.

Syntax

```
RotateWindows           # send active (frontmost) window to back
RotateWindows [-r]
      -r                # reverse rotation;
                        # bring bottom window to front
```

The **Format** command has a new attribute pair associated with the `-a` option. The option-attribute combinations are:

```
      -a L              # lock auto scrolling
      -a l              # unlock auto scrolling
```

There are four new commands: **ShowSelection**, **SaveOnClose**, **AddPane**, and **DeletePane**.

Syntax

```
ShowSelection           # place the selection in the desired window
                        # position
ShowSelection [-t | -b | -c | -n lines | -l line] [window]
      -t                # pin selection to top of window
      -b                # pin selection to bottom of window
      -c                # pin selection in center of window
      -n lines          # move selection to "lines" from top
```

```
      -l line           # move the line numbered "line" to the
                        # top of the window
```

This command scrolls the window, setting the selection to the desired position. If a window is not named, the default is the target window.

- If the selection is in the window at the time of execution of the above command, the window will not scroll. (See *Known Outstanding Bugs*.)

Syntax

```
SaveOnClose             # set window saving preference
SaveOnClose [-a | -d | -n] [window]
      -a                # always save window when closing
      -d                # default (ask Yes/No/Cancel)
      -n                # never save window when closing
```

Note: The -n option does not deactivate the Save menu item; the user may always save explicitly.

This command selects an automatic behavior: save (-a), do not save (-n), or ask whether to save when closing a window (-d). If a window is not named, the default is the target window.

The SaveOnClose option is not permanent; its effect will not persist beyond the closing of the window. Non-default "save on close" status is shown by one of the following icons in the upper right-hand corner of the window:

The left-hand one denotes "never save on close"; the right hand one denotes "always save on close."

If no option is given for this command, the behavior is to return a complete command line of the form: `SaveOnClose <option> <full window pathname>`, thus showing the SaveOnClose status of the named window.

- The next two commands, **AddPane** and **DeletePane**, are illustrated by an example that immediately follows the formal syntax presentations.

Syntax

```
AddPane                  # split the window into panes
AddPane [-p paneSpec] [-y ySplit | -x xSplit] [window]
      -p paneSpec        # choose a pane to split
      -y ySplit          # put a horizontal scroll bar at ySplit pixels
                         # from the top edge of the pane
      -x xSplit          # put a vertical scroll bar at xSplit pixels
                         # from the left edge of the pane
```

This command selects a pane of a window and then splits the pane as specified. `paneSpec` identifies a pane by a path that starts at the largest subdivision and works toward the smallest. `paneSpec` is a catenation of strings of the form "c*nn*" or "r*nn*" where the "c" and "r" stand respectively for row and column and "*nn*" is an appropriate ordinal (see examples below). If neither the `-y` nor the `-x` options are given, a set of `AddPane` commands, which would have produced the window as shown, is sent to standard output. If the `-p` option is omitted, the pane that is split is the *active* pane, i.e., the pane last written into. If a window is not named, the default is the target window.

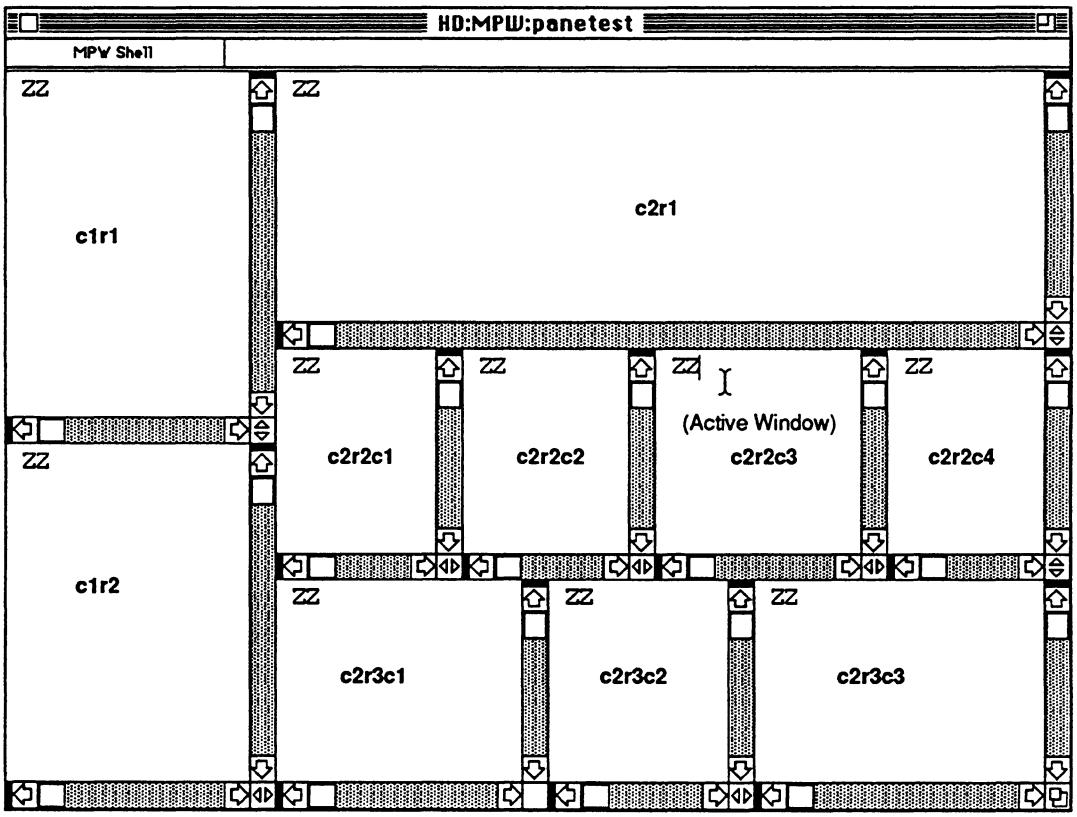
Syntax

```
DeletePane               # delete panes from the window
DeletePane [-p paneSpec | -a] [window]
      -p paneSpec        # choose a pane to delete
      -a                 # reset the window to have a single pane
```

This command selects a pane of a window and deletes it by removal of a scroll bar. `paneSpec` has the same definition and use as is given above in the description of `AddPane`. If the pane to be deleted is part of a row, the scroll bar on its right is deleted unless the pane is the rightmost pane of the row, in which case the scroll bar on the left is deleted. If the pane to be deleted is part of a column, the scroll bar at its bottom is deleted unless it is the lowest pane of the column, in which case the scroll bar at its top is deleted. If neither option `-p` or `-a` are given, the pane that is deleted is the *active* pane, i.e., the pane last written into. If a window is not named, the default is the target window.

Example:

- A multi-paned window. Each pane is labelled with its `paneSpec`.



The command `AddPane HD:MPW:panetest` displays the following commands, which would produce the above tiling when starting with an unpaned window:

```
addpane -p '' -x 159 hd:mpw:panetest
addpane -p c1 -y 219 hd:mpw:panetest
addpane -p c2 -y 163 hd:mpw:panetest
addpane -p c2r2 -y 135 hd:mpw:panetest
addpane -p c2r2 -x 110 hd:mpw:panetest
addpane -p c2r2c2 -x 113 hd:mpw:panetest
addpane -p c2r2c3 -x 136 hd:mpw:panetest
addpane -p c2r3 -x 161 hd:mpw:panetest
addpane -p c2r3c2 -x 120 hd:mpw:panetest
```

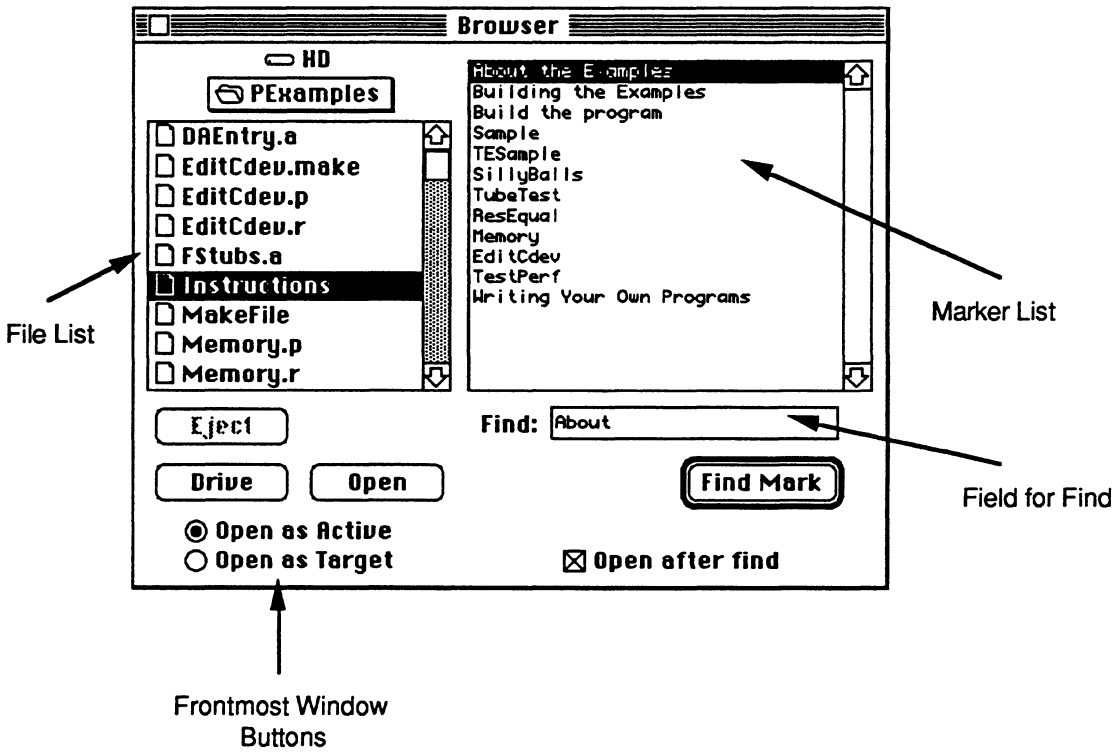
- *Note:* `addpane -p '' -x 159 <window>` is equivalent to `addpane -v 159 <window>`.

Browser

The Browser presents users with a standard file/directory list, similar to that of the "open" dialog box. If a file has been selected, a pane on the right shows all markers belonging to that file. Upon selecting and double-clicking a marker, the Shell will open the appropriate file and find the selected marker. If a marker is known, but the file containing it is unknown, a field for entering a string allows the user to search for a file that contains that string as a marker or a part thereof.

The Browser window is opened by executing the command `Browser` or choosing **Browse** from the Mark menu.

- Browser Window



File List

This pane shows all of the text files within the specified directory. Clicking on a filename causes the Marker List to display all of the markers within that file. Double-clicking on a filename opens the file. All aspects of file and directory selection work the same way as for the dialog box that appears when opening a file from an application, e.g. typing the initial letters of a file or directory name, use of the cursor control keys and of the latter in conjunction with the command key.

Marker List

This pane shows all of the markers within the selected file. Double-clicking on a marker opens the selected file to the selected marker.

- **See Also:** Mark command and the tools CMarker and ProcNames.

Field for Find

The user can type into this pane a string for comparison to all markers in all files in the current directory. Pressing Return or Enter causes the Browser to search for the marker that contains this string (even as a proper substring) and to select it. If the marker appears in more than one file, the first such appearance in the file list is selected. If the **Open after find** box has been checked, the file with the desired marker is automatically opened with the marked item selected.

Find Mark Button

This button will open the currently selected file to the currently selected marker if the current pane is the marker list (see Pane Selection below). Otherwise, if the current pane is the find field, it will begin searching for the text in the find field. Once one occurrence of a mark is found, this button changes to Find Next until the mark is deselected.

Frontmost Window Buttons

These radio buttons determine whether the window, when opened, is to be the frontmost window (Open as Active), or the window immediately behind the Browser window (Open as Target).

Pane Selection

The currently selected pane is indicated as follows: for File List and Marker List by a heavy border, for Field for Find by the blinking of the cursor. The default selection is **Field for Find**. The selection can be advanced, rotating among the three panes, by pressing Tab. When either the File List or Marker List pane is selected, typing the initial letter or letters of a name will select an item in the same manner as is done in the conventional dialogue windows for opening files.

Projector

◆ *Note:* A tutorial on Projector can be found as Appendix G of these Release Notes.

A number of improvements have been made to Projector. This includes correction of some deficiencies in the Commando interfaces, provision for checkout from and checkin to an NFS (UNIX Network File System) site, and the set of changes that are detailed below.

Cancel Checkout

To reduce the probability of an erroneous cancellation of a checkout, an alert box stating the identity of the owner of a checked out revision will appear when any user who is not the owner starts to cancel the checkout. When a checkout is cancelled, Projector will change the file from modifiable to read-only provided the file is to be found in the current checkout directory. If the file is not found there, an alert box with a warning message will appear. Cancel Checkout is now logged; instances of cancellation can be examined by use of the command `ProjectInfo` with the option `-log`.

AppleShare Use

Projector now supports multiple simultaneous reads from a Projector database. Thus, several users can simultaneously check files out without incurring delay. If, however, any user is writing to the database, other users are temporarily locked out. Conversely, a user wishing to write to the database is temporarily locked out if any other users are reading or writing.

Time-outs

If a user is temporarily locked out under the circumstances described above, a "retrying..." dialog box is displayed. The user has the option to cancel. Formerly, cancellation took place automatically after 80 retries.

ModifyReadOnly

More than one file name can now be written in the command line of a single ModifyReadOnly command.

NameRevisions, DeleteNames

The default for NameRevisions and DeleteNames has been changed from private to public. This required the introduction of a new option: `-private`. The old option, `-public`, is being kept so as not to invalidate old scripts. The behavior then is:

1. `-public` will work as before.
2. The new option, `-private`, must be used to define private names.
3. Not specifying an option gives the default value: `-public`.
4. Using the `-b` option (both) with NameRevisions will work as follows: when listing names (no files specified), both public and private names will be listed; when creating a name, `-b` will default to public.

OrphanFiles

OrphanFiles no longer changes the modification date of the file.

Verify

This feature was added to Projector after a number of experiences with loss of data from rotating magnetic storage. It is activated for Checkin and Checkout by marking a check box labelled "Verify" in the respective window, or by using the option `-verify` in the respective command line. The effects are as follows:

For Checkout: The file is checked out out normally and checked out out a second time as a temporary file. The data forks of the two files are then compared. If they are equal, the temporary file is deleted. If they are unequal, the temporary file is saved, an error message is printed to `StdErr`, and a status of 2 is returned.

For Checkin: The file is checked in, and then checked out as a temporary file. The data forks of the temporary file and the copy that had been checked in are compared. If they are equal, the temporary file is deleted. If they are unequal, the temporary file is saved, an error message is printed to `StdErr`, and a status of 2 is returned.

If the `-p` option is combined with the `-verify` option, a message will be emitted saying that the file has been verified

Warning Messages (-newer and -update)

When executing a `CheckOut` command with either of the options `-newer` or `-update`, various circumstances may prevent the checkout of one or more files. Warning messages are now issued for each such file, indicating the reason for the failure. The message formats are:

```
NOT checked out: Your file <filename> is a modified read-only file.
NOT checked out: Your file <filename> is a modifiable file on a branch.
NOT checked out: Your file <filename> is on a branch.
NOT checked out: Your file <filename> belongs to another project.
NOT checked out: Your file <filename> has no ckid resource.
WARNING: Your file <filename> is modifiable but the checkout was
      cancelled.
WARNING: Your file <filename> is not part of the project.
```

The first of the above warnings is issued only if the modified read-only file is not the latest revision.

The last of these is possibly not self-explanatory. It indicates that a `DeleteRevisions` command completely removed all revisions of the file from the Projector database. Then, a file bearing the same name as the file in question was checked in.

In the event that the CheckOut window is used with the Select Newer button (with or without the Option key), and one of the above circumstances occurs, a dialog box will be displayed with the message:

"WARNING: Some files have not been selected because they are not read-only, not part of the project or have some other problem. Use the 'checkout -newer' command for more detailed information."

Known Outstanding Bugs

- Editor performance suffers when files have an excessive number of marks (of the order of 100 to 200, depending on the speed of the cpu).
- (System 7) File aliases are not fully supported. They are currently usable only with the `Open` command.
- When using `ShowSelection`, if the selection is in the window at the time of execution the window will not scroll.
- Under System 7, tools that attempt to bring up the Standard File dialog in the background will bring up an empty window and totally freeze the computer so that only a manual restart is possible.

Bug Fixes

General

- The MPW Shell no longer hangs as a consequence of executing the `Beep` command while running in the background.
- The Shell no longer limits the stack of a tool to 200K.
- When using the Apple Extended Keyboard and Multifinder, the Cut, Copy, and Paste keys (F1–F3) now always update the clipboard when switching between applications.
- The `Files` command now correctly calculates the sizes of non-boot volumes when using the `-i`, `-x`, and `-l` options.
- The `Files` command formerly did not in all cases print out fill file names when the `-f` and `-n` options were used in conjunction with `-x` or `-l`. This bug has been fixed.

Projector

- Output from the `NameRevisions` command can now be interrupted by use of Command-Period.
- Formerly, `CancelCheckout`, when applied to a branch, would under certain circumstances cancel the wrong revision. This no longer occurs.
- A bug, which could cause database corruption under near-full-disk situations, has been fixed.

MPW 3.2 Tools/Scripts
Release Notes

Tools

Several new tools appear in this release.

- **StreamEdit** is a scriptable text editor similar to the editor *sed* that is found in UNIX®.
- **CMarker** generates markers at function definitions in C++/ANSI C source files.
- **Get,** developed for "411" Help, is available for general use. See the MPW 3.2 "411" Help Release Notes for information about "411".

A number of improvements have been made to existing tools.

- **Choose** has two new options for prompting for passwords.
- **Compare** has had its line number limitation raised from 9999 to 65535.
- **DumpObj** has a slightly changed **-m** option. The argument can now be either a module name or an entry point name. If it is the latter, the meaning is to dump the module that contains the named entry point. Any number of instances this option may now appear in a command line.
- **FileDiv** has been enhanced to give the user the option of considering its input file as a stream of bytes.
- The option **-sort** has been added to **GetListItem.**
- The performance of **Search** has improved as the result of search algorithm and buffering changes. Four new options have been added.
- **Link** has a new data initialization routine to reduce the size of the compressed data image in the link output. Appropriate changes have been made to DumpCode for conformity with this change in Link. Options have been added to Link, one to accommodate a new run-time architecture ("32-Bit Everything"), the other to cause automatic generation of branch islands.
- **Link** and **Lib,** upon failure, will set that date of the output file to numeric zero (1 January 1904).

- **Make** has several new features. The default rule mechanism has been extended so that it may identify optional additional dependencies in addition to the single dependency permitted by the current mechanism. A default build rule has been added for C++. A new predefined variable, {Deps} stands for all of the dependencies of a target. A new predefined target, $OutOf Date can be used to force other targets to be rebuilt. A new option, **-y**, provides a limited form of the verbose option output.
- **ProcNames** has been enhanced to generate MPW Shell "mark" commands that set markers on all procedures and functions in an Object Pascal file. It has also been enhanced to process conditional compilation directives.
- **PasRef** has been enhanced to process conditional compilation directives. The maximum number of symbols it can handle has been raised from 5000 to 6000.
- **ResEqual** now looks at resource attribute flags.

△ A bug has been reported in **DumpCode.** Information will not be dumped correctly from a module that has been compiled and linked as "model far" and that contains any A5-relocatable information.

A partial work-around for this is to use the "-ri" option, which inhibits the dumping of relocation information. This will, at least, permit correct dumping of code. △

Backup

A bug that has now been fixed prevented the creation of inner folders when the when the `-t` option was used to limit recursive processing (`-r`) to files of a specific type.

Choose

The following options, that prompt for "secure" passwords with a dialog box, have been added:

```
-askpw   #prompt for the server password.  Illegal in conjunction
           with -pw or -guest.

-askvp   # prompt for the volume password.  Illegal in conjunction
           with -vp or -guest.
```

CMarker

CMarker reads the specified C++ / ANSI C source file(s), syntax checks them and generates appropriate "Open" and "Mark" MPW commands, which, when executed, will mark the source file(s) at each function definition with the marker name being the name of the function. Its purpose is to aid in the marking of source files for use with the MPW "marker browser " capability. CMarker contains a full ANSI C preprocessor and provides options to mark include files, generate source listings (with or without showing macro expansions), run the preprocessor only, flag anachronisms, and syntax check C++ / ANSI C with or without Apple extensions.

(See Appendix B for the CMarker manual.)

Commando

The following bug has been fixed: conditionals in Commando resources were not always evaluated correctly when a set of dependencies was not a simple tree but a directed acyclic graph. An illustration of this would be two entries, B and C, both dependent on A, with D dependent on the expression (B or C).

Compare

The maximum number of lines in the files being compared has been raised. The limitation section of the manual should be amended to read: "The maximum number of lines in the text files read by Compare should be less than 65535."

DumpCode

DumpCode now stops disassembling at the end of the resource (rather than skidding to a halt a few bytes beyond). It recognizes and dumps the new data initialization format used by Link.

DumpObj

The `-mods` option prints a summary of the contents of an object file, including the name, size, scope and segment of each module and entry point. This means you can find out what's in an object file without dumping the entire file.

More information is being dumped (flags are printed in English, logical addresses are disassembled, and so forth).

Type data fixups are now printed correctly.

FileDiv

FileDiv now allows an input file to be viewed as containing an arbitrary byte stream in its data fork.

(See Appendix F for changes to the FileDiv manual.)

Get

Get is a tool for retrieving information from a data base indexed by a BTree. It will not only retrieve information, but will also create and update the index file when required.

Get is heavily oriented to the needs of the menu-driven "411". Direct calls to it could be for the purpose of retrieving information from the "411" database in a different manner from that provided in "411", or for accessing a database that is totally independent of "411".

Information on how to construct database files can be found in the section entitled 'Adding your own help to "411"' in the MPW 3.2 "411" Help Release Notes.

(See Appendix C for the Get manual.)

GetListItem

The option **-sort** has been added. Use of this option will cause the list in the dialogue box to appear in sorted order.

Lib

Lib now supports the `-mf` option to allocate MultiFinder temporary memory when the MPW Shell heap runs low. See the documentation (and warnings) about `-mf` in the Link manual page.

Link

Data Initialization

A new data initialization compression routine was added to Link to provide a more compact representation of the data image within link output. The routine originated from the need for better compression of C++ VTables. An associated _DATAINIT module exists within the 3.2 alpha Runtime.o library to expand the data image into the proper below-A5 world. The new routine may reduce the size of the %A5Init segment by up to 33%. Link will revert to using the original data initialization compression routine when older libraries are linked.

New Options

Two new options deal with removal of various 32K size limitations. For technical explanation, see the MPW 3.2 Run-Time Architecture Release Notes.

Branch Islands

A new option has been defined to cause the automatic generation of branch islands to remove the 32K limit on segment size. The syntax is:

```
-br on      # generate branch islands where needed
-br off     # do not generate branch islands (default)
```

◆ The `-br on` option should not be used simultaneously with the `-model far` option.

△ **Important** Because of an outstanding bug, attempting to use the `-br on` option simultaneously with the `-sn` option may cause a crash. △

"32-Bit Everything"

A new option has been defined to accommodate the "32-Bit Everything" method of removing the 32K limit on segment size, jump table size, and the size of the global data area. They are:

```
-model near       # the default
-model far
```

If any of the code being linked was compiled with a "far" option, it is necessary to link with the option `-model far`.

New Keywords (-opt option)

The `-opt` option accepts two additional keywords:

`-opt names`	SelectorProcs are modules generated by the linker to be used at run-time for Object Pascal method dispatching. This keyword causes MacsBug symbols to be appended to SelectorProc modules so that the selector names are visible in MacsBug disassemblies. The MacsBug symbols take up space in the application, both on disk and in memory (e.g. 9K or more for a medium MacApp application).
`-opt info`	Write method table optimization information to diagnostics. This information was previously written to diagnostics using the '-p' option and was moved to the 'Info' subarg to reduce the size of the linker diagnostics when optimizing.

Object Pascal Optimization

Jump table entries for monomorphic methods are now being stripped. This is an intentional step of the optimizer, although it may present problems when trying to call an Object Pascal method from a C++ program. To work around the dispatching problem, use the `NoBypass` subargument to `-opt` to bypass monomorphic method optimization when optimizing.

Symbolics

The format of Sym files has changed to support Object Pascal and FORTRAN. Use the 1.3 SADE with the MPW 3.2 Linker (you can't mix old and new Sym files or SADEs — please delete your existing Sym files). Contact DTS for the details of the Sym file and OMF changes.

Miscellaneous

The linker now returns an error when the size of the jump table has reached the maximum number of jump table entries.

Previously, when the `-ad` option was used, the linker would align every data module including the main data module, if one existed. This was unacceptable since MPW Pascal makes A5-relative references to data in the main data module assuming that it is located immediately below A5. The `-ad` option will now correctly align data while leaving the main data module immediately below A5.

There is now no limit to the size of global data that is compressed into the %A5Init data initialization segment. There was previously a limit of 32K.

When linking using the `-mf` option, if memory conditions continue to be tight, the flush command may be used to recover additional space from the shell's cache.

Various cosmetic changes have been made to the `-l` and `-map` output.

Make

The new version of Make for MPW 3.2 includes several new features.

The default rule mechanism has been extended so it may identify optional additional dependencies in addition to the single dependency permitted by the current mechanism. This new capability has immediate applications to MacApp builds.

A default build rule has been added for C++ files, which are recognized by the suffix `.cp`.

Make supports a new predefined variable, {Deps}, which stands for all of the dependencies of a target.

Make also supports a new predefined target, $OutOfDate, which can be used to force other targets to be rebuilt.

A new option, **-y**, has been provided. This option tells why build rules were emitted, but in a less verbose form than the existing **-v** option. Namely, it does not emit messages about up-to-date targets.

Default Rules Extension

Default rules formerly had the following form:

```
.extension1 ƒ     .extension2
      <build rules>
```

This syntax has been extended as follows:

```
.extension1 ƒ     .extension2   [other dependencies ...]
      <build rules>
```

The other dependencies can be more file extensions and/or fixed file names. These dependencies are considered secondary or optional dependencies as is explained below.

The first extension on the right hand side of a default rule is treated as the **primary dependency** (or trigger dependency), that is, the file name created with this extension (i.e., "{depdir}{default}.extension2") must be valid in order for the default rule to be triggered or applied. To be *valid* a file name must either appear in the makefile or exist in the file system (or lead to a valid file name by further recursive applications of default rules). The existence or non-existence of files specified by secondary dependencies will have no effect on whether the rule is triggered, thus default rules are triggered just as they were before.

Secondary dependencies are optional; that is, none are required, and the indicated file (or files) need not be valid for the default rule to be applied. Secondary dependencies are only processed when a default rule is triggered by its primary dependency. When the secondary dependency is a fixed file name, a dependency will be added if the secondary dependency refers to a valid file name. Similarly, when the secondary dependency is a file extension (e.g., ".extension3") a dependency will be added if the file name implied by this extension (i.e., "{depdir}{default}.extension3") refers to a valid file name. If a file name specified by a secondary dependency is not valid then no dependency is added and the default rule is processed as usual.

Applications. This extension will be useful for MacApp, where typically a source file consists of a file with the interface and an include with the implementation. Now the dependency of the object file on both source files can be stated in a single rule, such as the one below:

```
.p.o  ƒ     .p    .p.incl
      <normal Pascal build rule>
```

"Deps" Variable

The predefined "{Deps}" variable may be used in a target's build rules and represents all of the (first-level) dependencies of the target (as opposed to {NewerDeps} which represents only those dependencies which are newer than the target).

The {Deps} variable can be used to write a default rule for links or application builds where all of the dependency files will be linked together, such as:

```
.     ƒ     .o
      {Link}  {LinkOptions}  {Deps}  -o {TargDir}{Default}
```

"$OutOfDate" Target

The predefined "$OutOfDate" target is an artificial target which is always out of date and never needs to be rebuilt. One can force a target to be rebuilt by making it depend on $OutOfDate.

For example:

```
ForceRebuild =          # default variable definition (NOP)
foo         ƒ     {ForceRebuild}
      <build foo>
```

If Make is invoked normally the target "foo" will not be rebuilt; however, it will be rebuilt if invoked with the following command line:

```
Make  -d  ForceRebuild=$OutOfDate
```

This command line has the effect of overriding the vacuous definition of "ForceRebuild" in the makefile while giving foo a dependency on $OutOfDate which forces it to be rebuilt.

PasMat

The following bug, pertaining to the `j` formatting directive, has been fixed:

```
j=<width>[±]/<col1>[sd]/<col2>c
```

did not handle the case where `<col2>` could be 1.

PasRef

Options have been added to permit processing of conditional compilation directives. The limitation on the number of symbols has been raised from 5000 to 6000.

(See Appendix E for changes to the PasRef manual.)

ProcNames

ProcNames can be directed to generate MPW Shell "mark" commands that place markers on all procedures and functions in an Object Pascal file.

Options have been added to permit processing of conditional compilation directives.

(See Appendix D for changes to the ProcNames manual.)

ResEqual

ResEqual now compares resource attribute flags for equality and reports any differences found.

Rez

Avoid splitting expressions with #if or #elif commands, for example:

```
type 'TEST' {
      int;
};

#define big 5

resource 'TEST' (128) {
      10    // this will not work because the expression is split
#if big
      * 45
#endif
};
```

- Attention is directed to the material in the MPW 3.0 Reference Manual with respect to the arguments of the `-c` and `-t` options. These arguments are Rez expressions. If a special character within the expression requires that it be enclosed in single quotes, it is necessary to surround the quoted expression with double quotes. E.g. `-c " '§§§' "`.

Search

The following options have been added:

```
-b    #break File/Line output text into two lines

-nf   # write error message if pattern not found

-ns   # return 0 when pattern not found

-sf   # stop the search at the first successful match
```

Improvements in the search algorithm and in the buffering have greatly enhanced the performance.

Sort

A bug in Sort prevented the sorting of field expressions that include column offsets or counts. For example, a call of Sort with the option

```
-f 1.3+4
```

would fail to do any actual sorting. This bug has been fixed.

StreamEdit

StreamEdit is an non-interactive text editor similar in function to the Unix® tool *sed*. Providing scriptable text matching and editing operations, it is useful for making repetitive changes to files, for extracting information from text files, or as a filter.

StreamEdit takes a script and a set of input files (or standard input, if no input files are specified) and applies each statement in the script to each line of input, writing the output to standard output or the specified output file.

(See Appendix A for the StreamEdit manual.)

Scripts

CompareFiles

The CompareFiles command now has two more options for specifying screen size:

```
-TwoPage    # screen size for Apple Two-Page Monochrome Monitor
-Portrait   # screen size for Apple Macintosh Portrait Display
```

CreateMake

Enhancements to CreateMake are:

- A check box for providing symbolic information to the SADE debugger. This will apply the **-sym on** option to the generated command lines for compilation and linking. The check box is dimmed for desk accessories, because the method for building them produced by CreateMake is note compatible with SADE.
- Radio buttons for the options **-mc68020**, **-mc68881**, and **-elems881**.
- Provision for the building of objects of a new type: SIOW application. This type is described in the MPW 3.2 SIOW Release Note.

Other changes are:

- CreateMake now creates correct Makefiles for building cdevs and most stand-alone code resources. Desk accessories remain a problem since there are several different methods for building the header at the start of a desk accessory. CreateMake will produce a correct makefile for one of the methods used in the Memory DA examples found in the CExamples and PExamples folders. This build technique is the same as that found in MPW 3.0, but is not compatible with SADE. For information see the Instructions file in the CExamples or PExamples folder. In conclusion, if you are building a desk accessory, you probably shouldn't be using CreateMake to produce the makefile.
- CreateMake recognizes that the functions of the libraries CRuntime.o and CInterfaces.o have now been absorbed into Runtime.o and Interfaces.o; it therefore no longer puts CRuntime.o and CInterfaces.o into the library list for linking.

- The compilation and linking command lines generated by CreateMake no longer have the **-w** option; warnings will be issued. The only exception is that the link command line for a tool has the **-d** option, thus suppressing the duplicate definition warnings that would normally be issued because of the use of Stubs.o.
- If any source file is written in Object Pascal or C++ (filenames with the extensions `.p` or `.cp`), the line `#"{Libraries}"ObjLib.o` will appear in the link.

MPW 3.2 Run-Time Architecture Enhancements
Release Notes

Introduction

The original execution environment of the Macintosh was designed for a machine with 128K of RAM and a 68000 CPU. In order to cope with a memory this small, dynamic relocation of almost everything was a necessity. It had to be possible to move segments of code, to purge them when not needed, and to reload them elsewhere than their original location when needed again. The initial architecture restricted the size of each of the following to 32K: code segments, the global data area, and the *jump table* (described below). This execution environment is considered to be an extremely efficient one within the stated constraints.

Throughout the history of MPW, various enhancements to MPW tools have been made with the purpose removing one or another of these 32K restrictions. The following sections of these Release Notes show how to apply the available alternatives:

- Direct compiler generation of A5-relative data references with offsets exceeding 32K.
- Direct function calls within a code segment greater than 32K by use of the 68020 PC-relative branch with 32-bit displacement (C compiler, 68020 only).
- Making function calls within a code segment greater than 32K by using "branch islands."
- A Link option providing for extension of the jump table to more than 32K by utilizing part of the application globals space.
- The "32-Bit Everything" solution, which extends application globals, code segments, and the jump table to more than 32K each by performing address "fixups" at load time.

The remaining sections of these Notes describe the technology of the "32-Bit Everything" solution in detail, present a number of essential caveats, and describe a new run-time library and interface that is directed to applications (and debuggers) which might be explicitly concerned with the "32-Bit Everything" Segment Loader patches.

Breaking the 32K limit

More than 32K of global data (all 680x0 machines)

The compiler option -m, available for both Pascal and C, causes generation of code sequences that yield A5-relative references (execution-time A5 value) with 32-bit offsets.

A typical sequence is:

```
      MOVEA.L     A5,A0
      ADDA.L      <32-bit offset>,A0
      WHATEVER    (A0)
Without the -m option, the code would be:
      WHATEVER    <16-bitoffset>(A5)
```

In either case, the offsets are supplied by the Linker.

Disadvantages: Code is larger and slower

Code segments greater than 32K (C compiler, 68020 and above)

The C compiler option -bigseg causes function calls within the same segment to be encoded with the 68020 BSR.L instruction, which is a PC-relative instruction with a 32-bit offset.

Disadvantage: Not available to 68000 machines.

Advantage: Useful for large single-segment code, e.g. XCMDs.

Branch islands—a universal technique for large code segments

A simple, effective technique for implementing PC-relative code-to-code references within a segment that exceeds 32K in size is to split the segment into two independently compiled modules each of which is under 32K, and to include as a sandwich between these two portions a small assembly-language module (a "branch island") that transmits calls between the two. In other words, the original call is modified to be a JSR to the branch island and the latter contains a BRA to the desired target. For usage information, see the **Link** section of the MPW 3.2 Tools/Scripts Release Notes.

Large jump tables

A Linker option, `-wrap`, makes it possible for jump tables to be larger than 32K by utilizing unused space in the global data area for jump table entries when the jump table space has been exhausted. This is particularly useful for MacApp programs. Because they make little demand on global data space, this option significantly increases the size of the applications that can be generated with MacApp.

Disadvantage: Of limited utility. At best, can double the jump table size. Not applicable if the global data area is filled with data.

"32-Bit Everything"

This is a complete and almost transparent method of removing all three limitations: on code segment size, jump table size, and the size of the global data area. The cost to the applications developer is a slight increase in code size and a slight increase in the time to load a code segment. The method is activated by using certain options when compiling and linking. The compiler options permit choice, on a compilation unit basis, of full 32-bit offsets for global data (`-Model farData`), full 32-bit offsets for code references (`-Model farCode`), or both (`-Model far`). Linking of modules compiled with any such combination is supported; the only requirement is that if any 32-bit option has been used with any of the compilation units, the linking must be done with the option `-Model far`. The increase in code size has three causes: each reference occupies two bytes of additional storage, and the Linker inserts into the executable file relocation information that is used when loading and about 1K of code that patches the Segment Loader . The increase in load time is caused by dynamic relocation of the 32-bit references during loading. Global data references and code references via the jump table are relocated by adding the load-time value of A5. Intra-segment code references are relocated by adding the load address of the segment.

Compiler and Linker options

The options are:

Compiler options (Object Pascal and C):

```
-Model near       # the default
-Model far
```

which respectively choose the stated model for both code and data,

```
-Model nearData   # again, the default
-Model farData
```

which chooses a model for data only, and

```
-Model nearCode    # default
-Model farCode
```

which chooses a model for code only.

Linker options:

```
-model near        # the default
-model far
```

If any code being linked was compiled (or assembled) with any "far" option, it is an error to attempt linking without specifying the option `-model far`.

Assembly language options and techniques

In assembly language, the use of a 32-bit reference for the target address of an instruction must be explicitly demanded by use of the absolute long address syntax `(xxx).L`, where "xxx" is a relocatable expression. Two further requirements are that the relevant operand symbol be imported, and that the option `-model far` be used for the assembly. The requirement that the symbol be imported means that the defining occurrence of the symbol must be in a different module from the instances of use as 32-bit references. Since the absolute long address syntax by definition specifies absolute operands, the use of this form with a relocatable symbol is an error unless the option `-model far` is invoked. The absence of the option `-model far` (the default) can be explicitly shown by use of the option `-model near`.

Global data references, references to code in the same segment, and references to code in a different segment all cause the assembler to produce similar records, records that tell the Linker that a 32-bit patch will be needed. The Linker observes whether the references are to code or data, and in the former case, whether the reference is within the same segment or not.

The following example illustrates the technique:

```
                MAIN
                IMPORT      STUFF             ; Symbols from other
                IMPORT      THERE             ; modules must be
                IMPORT      ELSEWHERE         ; imported.
                JSR         (THERE).L         ; Symbols are written
                JSR         (ELSEWHERE).L     ; using (xxx).L syntax.
                ADD.W       (STUFF).L,D0
                ENDMAIN

                PROC                          ; Note that THERE
                EXPORT      THERE             ; is in the MAIN
THERE           NOP                           ; segment.
                ENDPROC
```

```
                SEG         'SG1'           ; Note that ELSEWHERE
                PROC                        ; is in a different
                EXPORT      ELSEWHERE       ; segment.
ELSEWHERE       NOP
                ENDPROC
                PROC
                DATA
                EXPORT      STUFF
STUFF           DS          1
                ENDPROC
                END
```

Known problems

- The 32-Bit Everything mechanism patches the `_LoadSeg` trap. Therefore, if user code calls `_LoadSeg` directly or patches `_LoadSeg`, unpredictable behavior may occur.
- The `ObjLib.o` library does not support 32-Bit Everything. Because of this, code using Pascal Objects will not work with 32-Bit Everything. This does not apply to MacApp, whose library does support 32-Bit Everything.
- It is in general not correct to combine Object Pascal objects compiled respectively with differing choices of the *near* and *far* options. The Linker will consider this circumstance to be an error.
- The user should be aware that using the "far" option results in the creation of code which is not "pure." Because the code is modified when loaded (or reloaded), check sums become invalid and attempts to share code are likely to fail.

Technical details of "32-Bit Everything"

Recapitulation of 16-bit technology

In order to provide for relocatability of code segments, all code-to-code references within a segment are written as PC-relative, so that the code need have no knowledge of where it is loaded. This nominally restricts the size of segments to 32K because the PC-relative instructions on a 68000 use a 16-bit offset. Current ways of removing this restriction are not fully satisfactory, as a direct 32-bit branch is available only on the 68020 or higher.

References from code to global data appear as A5-relative addresses with negative offsets. The present method of dealing with more than 32K of global data is to generate 68000 instruction sequences that implement 32-bit A5-relative references. (The equivalent 68020 addressing mode is slow, and using it would introduce an undesirable processor restriction.)

Code references from one segment to another are performed as indirect references via the dynamically maintained jump table. Details of the jump table format are given in the Segment Loader chapter of *Inside Macintosh* (Vol. II, Ch. 2). The jump table is loaded at a small positive offset from A5, and references to it are encoded as A5-relative addresses with positive offsets. Each procedure or function which is an entry point to its segment is therefore represented in the code as an A5-relative address, this address being created by the Linker. The A5-relative addressing mode suggests that the jump table be limited to 32K. Other architectural considerations in fact prevent extension of this limit.

In order to help the reader understand the description of the new addressing mechanisms, a short summary of the current jump table mechanism follows. For more details, see the above-named chapter of *Inside Macintosh* . The jump table entries for the procedures of any given segment are contiguous. They appear in two alternate formats, depending on whether or not the segment has been loaded. The "unloaded" format contains the all the information the segment loader needs to construct an absolute jump instruction to the procedure. The information is: the segment number and the offset of the procedure from the start of its segment. A subroutine jump to the "unloaded" jump table entry results in a call of the Segment Loader to load the segment and to modify all jump table entries to the "loaded" format. In this latter format, the jump table entry holds, in the same eight bytes, the segment number and the previously-mentioned absolute jump instruction. When the segment is purged, the segment loader reverses the process, and returns that segment's jump table entries to the "unloaded" format. Each segment starts with a header, containing information that the Segment Loader needs in order to perform the switch between "loaded" and "unloaded" formats. This information is the offset of the first entry for that segment from the start of the jump table and the number of jump table entries belonging to the segment.

In summary, the system was designed with a nominal restriction to 16-bit addresses for several types of references, and the various methods of partial removal of this restriction are not considered adequate. What is wanted is a consistent, unified, backwards-compatible, and efficient way of removing the restriction.

The "32-Bit Everything" solution

The tricky part of removing the restriction to 16-bit addressing is to do it in such a way that the changes are almost transparent. If we refer to code with 16-bit addressing as code according to the "near" model, and code with 32-bit addressing as the "far" model, then one ground rule is that the Linker should happily accept a mixture of segments compiled with different models. Since the code making an intersegment reference cannot "know" the model of the addressee, the code must make the call via the jump table in a model-independent manner. This, of course, also requires that there be no change in the size of a jump table entry.

There are three types of references which must be considered: code to data, intra-segment code to code, and inter-segment code to code. Each of these is described in detail below. The basic methodology is the same for each: instead of having compilers generate PC-relative and A5-relative instructions, both having 16-bit offsets, have compilers instead generate instructions with 32-bit addresses, and relocate these addresses at load time by the segment load address or by the contents of A5, as is appropriate. Obviously the Segment Loader must change. The way in which this change has been implemented will be described later.

Code-to-data

If compilation and linking are performed with any option that specifies the "far" model for data, then all instructions which reference global data are generated with 32-bit absolute addresses. These addresses are the offsets of the data items relative to A5. The location of each such instruction making such a reference is stored in compressed form in an area called "A5 relocation information." The modified Segment Loader, using this information and the value of A5 at load time, relocates each such instruction during loading by adding the value of A5 to the 32-bit address field of the instruction. This provides a more efficient method than hitherto available for referencing a global data area that exceeds 32K.

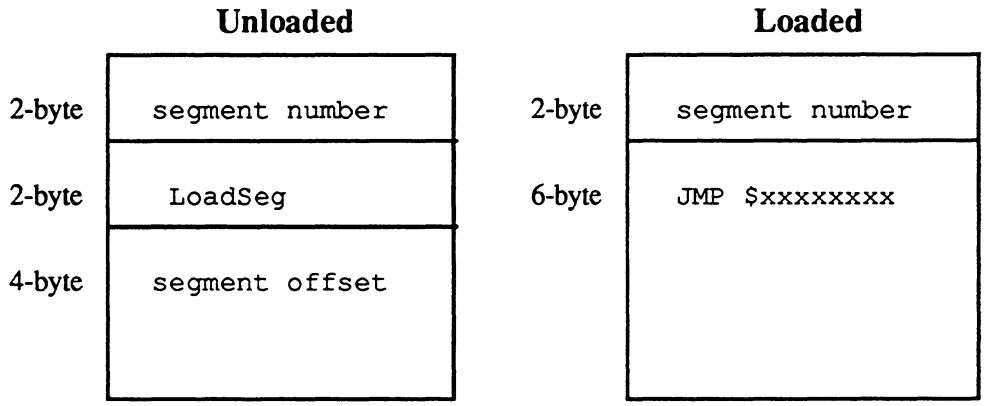
Code-to-code (intra-segment)

If compilation and linking are performed with any option that specifies the "far" model for code, then all instructions which make intra-segment code references are generated with 32-bit absolute addresses. These addresses are the byte offsets from the beginning of the segment to the referenced points. The location of each instruction making such a reference will be stored in compressed form in an area called "segment relocation information." The modified Segment Loader relocates each such instruction at load time by adding the load address of the segment to the 32-bit address field of the instruction. This permits a segment to be over 32K in length.

Code-to-code (inter-segment)

Compilation and linking with the "far" model causes the simultaneous removal of two previous restrictions: the limitation of the jump table and segment size to 32K bytes each. Because segments compiled under the "near" module may be linked with other segments that were compiled as "far," the jump table entries for "near" segments precede those for "far" ones. Thus, there still may be a portion of the jump table as large as 32K containing "near" entries before the start of the "far" entries. A segment compiled/linked as "far" may exceed 32K; therefore four bytes are required in a jump table entry to describe the offset of a function from the beginning of its segment. This requires a reorganization of the jump table from the "classic" (*Inside Macintosh*) structure. Because of the requirement simultaneously to support the "near" and "far" models, all jump table entries must be of the same size, and preferably of the same format. The new format follows:

- Jump table entry

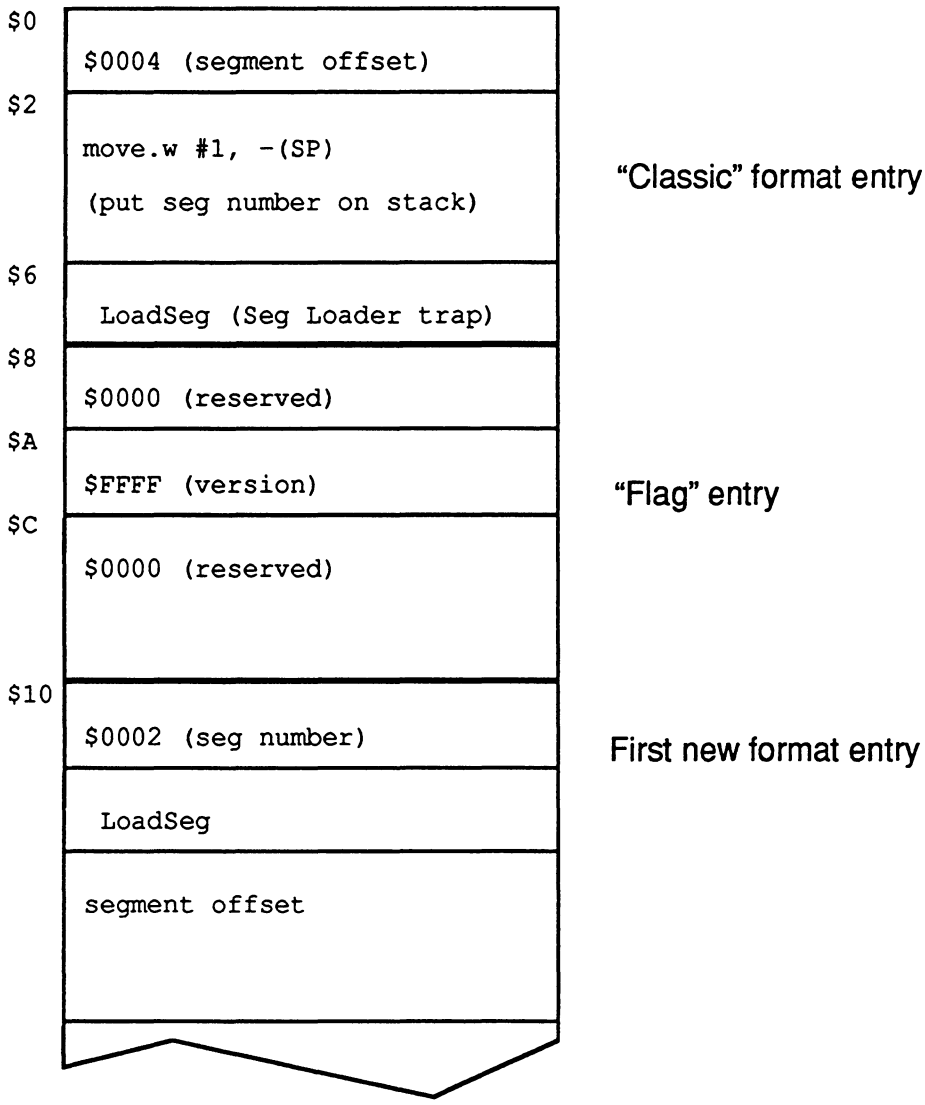


Only the entries for unloaded segments differ from the "classic" jump table entries documented in *Inside Macintosh*. By omitting the instruction that stacks the segment number (letting the Segment Loader fetch the segment number from the entry itself), two bytes are saved, making possible the recording of a 4-byte offset.

References to "far" jump table entries are handled in a similar way to global data references. JSR instructions with absolute addresses are generated, the addresses of these instructions are recorded as "A5 relocation information," and the (modified) Segment Loader adds the value of A5 to the address fields of the JSR instructions at load time.

Because MPW rides on top of an existing system, it was decided to create the modified Segment Loader by patching the one supplied by the system. The scenario is that, if the "far" model is requested, the Linker puts the code needed to do this patching into the application itself. The first entry in the jump table is a "classic" entry, understood by the standard Segment Loader, which points to the patch code. Execution of this code causes the desired modifications of the Loader. The remainder of the jump table is in the new format, this part being separated from the first entry by a "flag" entry of which the first word is reserved and must be zero, the second is a version field (nominally $FFFF), and the remaining two words are zero. The third jump table entry addresses the main entry point of the application. The following illustration shows the start of the jump table:

- Jump table structure



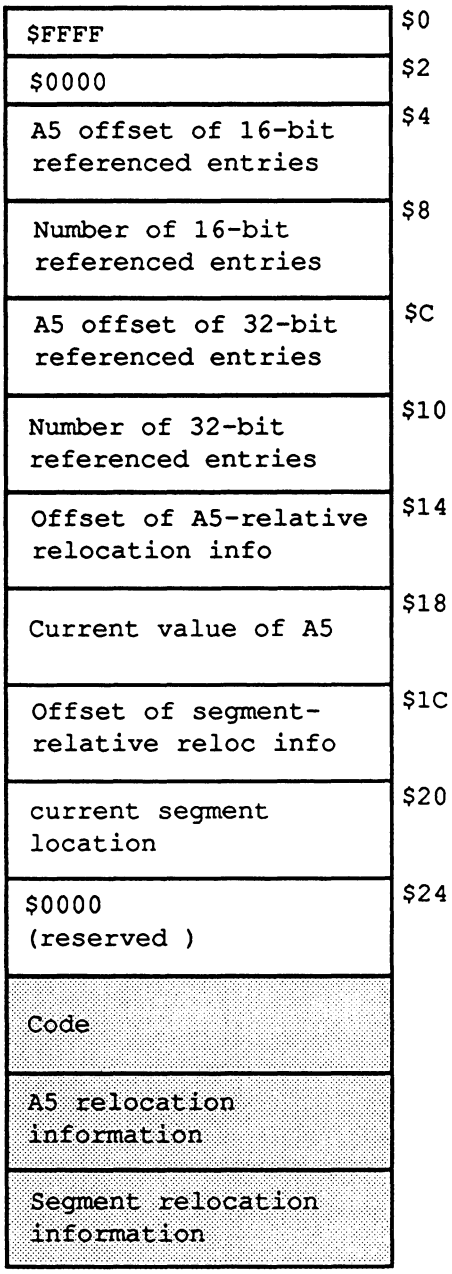
△ **Important** The format of the jump table for "32-bit Everything" is subject to change. Developers are cautioned against assuming any dependency on the format reported here. △

The standard Segment Loader loads 'CODE' resource 0 (jump table) and 'CODE' resource 1 in the usual manner. (The first entry in the jump table is a "classic" entry, understood by the unmodified Segment Loader.) The contents of code(1), when executed, patches the Segment Loader so that it can load and unload 32-bit segments. Code(1) also patches `Chain`, `Launch`, and `ExitToShell` so that, when executed, they unpatch the Segment Loader and then execute normally. Finally, Code(1) calls the third jump table entry. Because the third entry has not yet been loaded, loading is now performed by the modified Segment Loader, which, after verifying that the "flag" entry in the jump table is correct with respect to the version field in the segment header, updates all PC-relative and A5-relative references while loading the code. If a segment has been moved, or A5 has changed since the segment was loaded, then the jump table entries for the segment revert to the "unloaded" format, and the modified segment loader will add to the addresses in the code that need updating the respective changes in the load address and in the A5 contents.

32-bit segment structure

As stated above, "Classic" segments have a 4-byte header, the first two bytes being the offset from the beginning of the jump table of the entry for the first routine in the segment, and the second two bytes being the number of jump table entries for the segment. 32-bit segments have a considerably longer header and contain relocation information. That a segment is of the 32-bit persuasion can be determined from its first word, which matches the "version" field in the second jump table entry, namely $FFFF. The second word is $0000. The next longword is the byte offset from A5 of the first "Classic" or near jump table entry. This is followed in turn by longwords giving the number of near entries, the byte offset from A5 of the first 32-bit jump table entry, and the number of 32-bit entries. The next four longwords contain, respectively, the offset from the segment start of the relocation information for A5-relative references, the current A5 value used for relocating these references, the offset from the segment start of the relocation information for segment-relative references, and the segment load address used for relocating these references. Finally, there is a longword containing zero that is reserved for future use. Following this header is the code, the A5 relocation information, and the segment relocation information. This is shown in the following diagram (The order of the shaded items is subject to change.):

- Segment structure



Relocation information format

The relocation information is simply a consecutive list of offsets between longwords that need to be relocated at load time, beginning with the offset of the first such longword from the start of the segment. Some data compression has been used in recording this information. Because instructions start at even addresses, it suffices to record the offset values divided by two. In the table below, the various encodings are shown as bit strings. The portions denoted by "bbb..." give, when doubled, the desired offset values.

- Relocation information

relocation item	interpretation
00000000 00000000	end of relocation information
0bbbbbbb	offsets between $02 and $FE
1bbbbbbb bbbbbbbb	offsets between $0100 and $FFFE
00000000 1bbbbbbb bbbbbbbb bbbbbbbb bbbbbbbb	offsets between $00010000 and $FFFFFFFE

Signal handling in MPW tools

The MPW 3.2 A1 Release Notes stated a requirement that tools provide their own handling of the SIGINT signal (command-period) if any "far" option had been used in building the tool. This is no longer necessary, because the Shell will now remove the Segment Loader patches after termination of the tool.

Run-time support

△ **Important** Developers are cautioned that the material in the remainder of this document will probably change in MPW versions after MPW 3.2 △

Overview

This section discusses a set of interfaces and libraries for working with the "32-Bit Everything" run-time environment. Addressed particularly are applications and/or debuggers which use the _Launch or _Chain traps, or patch the _LoadSeg trap.

Files associated with this section are:

`{Libraries}RTLib.o`	Library that implements the interface;
`{CIncludes}RTLib.h`	C and C++ declarations;
`{PInterfaces}RTLib.p`	Pascal declarations.

Run-time interface

There are two kinds of programs that utilize the run-time interface: applications which need knowledge of the environment in which they are executing and resident debuggers which need to know about *another* application's environment.

◆ *Note:* The calls and data structures to follow are shown in C. Pascal equivalents are to be found in `RTLib.p`.

The "32-Bit Everything" run-time interface consists of a single procedure call:

```
pascal OSErr Runtime (RTPB* runtime_parms);
```

The operation to be performed is determined by the value of the first member of the structure `RTPB`. `RTPB` has the form:

```
struct RTPB {
   short   fOperation;
   void*   fA5;
   union {
           RTGetVersionParam   fVersionParam;
           RTGetJTAddrParam    fJTAddrParam;
           RTSetSegLoadParam   fSegLoadParam;
   }  fRTParams;
};
typedef struct RTPB RTPB;
```

The first member, `fOperation`, can have any of the following values:

`kRTGetVersion`	`kRTGetVersionA5`
`kRTGetJTAddress`	`kRTGetJTAddressA5`
`kRTSetPreLoad`	`kRTSetPreLoadA5`
`kRTSetSegLoadErr`	`kRTSetPreLoad`
`kRTSetPostLoad`	`kRTSetPostLoadA5`
`kRTSetPreUnload`	`kRTSetPreUnloadA5`
`kRtPreLaunch`	
`kRTPostLaunch`	

Those operations whose name ends in "A5" require a value for the second member, `fA5`, which is the address of an "A5-world." The similarly-named operation without the "A5" uses the current value of A5 as this address.

The third member, `fRTParams`, is a parameter block consisting of one of three structures holding parameters for the appropriate operation. The details for each structure are included with operation's description.

Possible errors returned from Runtime include:

Error	Description
`eRtNoErr`	(success)
`eRTInvalidOP`	Invalid operation
`eRTBadVersion`	Not 32-bit everything run-time
`eRTInvalidJTPtr`	Invalid jump table pointer

Run-time operations

`kRTGetVersion, kRTGetVersionA5`

Return the run-time version number for the current (or specified) A5-world.
The `fRTParams` structure used with these operations is:

```
struct RTGetVersionParam {
   unsigned short fVersion;
};
typedef struct RTGetVersionParam RTGetVersionParam;
```

`fA5` is used only for `RTGetVersionA5` to specify the A5-world. This field is not used for `RTGetVersion`.

`fVersion` holds the returned version number. Current version numbers are:

Version No.	Description
$0000	'Classic' world.
$FFFF	32-Bit Everything world.

`kRTGetJTAddress, kRTGetJTAddressA5`

Return the address of the code that the specified jump table entry points to for the current (or specified) A5-world.

The `fRTParams` structure used with these operations is:

```
struct RTGetJTAddrParam {
   void*   fJTAddr;
   void*   fCodeAddr;
};
typedef struct RTGetJTAddrParam RTGetJTAddrParam;
```

`fJTAddr` points to a given jump table entry. Pointing to a non-valid jump table entry will return an unpredictable result.

`fA5` is used only for `kRTGetJTAddressA5` to specify the A5-world. This field is not used for `kRTGetJTAddress`.

The code address is returned in `fCodeAddr`. Zero is returned if the segment is not loaded.

Segment Loader hooks

The operations `kRTSetPreLoad`, `kRTSetSegLoadErr`, `kRTSetPostLoad`, `kRTSetPreUnload`, and the operations of the same names with a trailing "A5" enable programs to acquire control in the case of error, or at pre-LoadSeg, post-LoadSeg, and pre-UnloadSeg times. They do this by specifying, for each of the above cases, a user handler to replace the (dummy) user handler that exists in the patched Segment Loader.

The fRTParams structure used with these operations is:

```
struct RTSetSegLoadParam {
   SegLoadHdlrPtr              fUserHdlr;
   SegLoadHdlrPtr              fOldUserHdlr;
};
typedef struct RTSetSegLoadParam RTSetSegLoadParam;
```

`fUserHdlr` is a pointer to the user handler to be called at the time indicated by the operation. A pointer to the replaced user-handler is returned in `fOldUserHdlr`. This pointer can be used at a later time to reinstall the original handler.

A user handler is defined as follows:

```
typedef pascal short (*SegLoadHdlrPtr)(RTState* state);
```

User-handlers may return a result code of type `short`. For now the result code is ignored by the segment loader except in the case of the error-handler (see Take an action).

△ **Important** User-handlers *must* be defined within a segment that will be loaded in memory when the handler is invoked. This will most commonly be the main segment. Also, a user-handler should not make any calls to functions within an unloaded segment because this may result in a system crash. △

`fA5` is used only for A5 operations and contains the value of register A5 for the specified A5-world.

The `RTState` structure is used to pass information about segment loader operations to the user-handler. It has the form:

```
struct RTState {
   unsigned
     short    fVersion;          // run-time version
   void*      fSP;               // SP: &-of user return
                                 // address
   void*      fJTAddr;           // PC: &-of jump table entry
   long       fRegisters[15];    // registers D0-D7 and
                                 // A0-A6
   short      fSegNo;            // segment number
   ResType    fSegType;          // segment type (normally
                                 // 'CODE')
   long       fSegSize;          // segment size
   Boolean    fSegInCore;        // true if segment is in
                                 // memory
   Boolean    fReserved1;        // (reserved for future use)
   OSErr      fOSErr;            // error number
   long       fReserved2;        // (reserved for future use)
};
typedef struct RTState RTState;
```

`fVersion` is the version number of the current "32-Bit Everything" run-time world.

`fSP` is the current stack pointer when either `_LoadSeg` or `UnloadSeg` was executed. In the case of `_LoadSeg`, if the jump table entry was reached via `JSR`, `fSP` is a pointer to the user return address. The value at this SP may be modified within the error handler to change the return address if a "continue" action is taken (see Take an action). This is not recommended since there may be no user return address on the stack (see discussion below on stack contents). In the case of `UnloadSeg`, `fSP` points to the return address from the `UnloadSeg` call.

`fJTAddr` points to the jump table entry called by the user code prior to the `_LoadSeg` call. In the case of `UnloadSeg` it is the jump table entry pointer passed to `UnloadSeg`. This field may be edited by the error handler to provide a different point of re-entrance when issuing the "retry" action (see Take an action).

It is important in the "32-Bit Everything" world that **nothing** be assumed about the actual layout of the jump table entry, since the format may change in new versions.

`fRegisters` is an array of longs which contains the register values at the time `_LoadSeg` was called. The registers are saved in the order D0 through D7, then A0 through A6.

`fSegNo` and `fSegType` contain the segment's resource type and id. `fSegType` will normally be 'CODE', but this may change in the future.

`fSegSize` contains the segment's size.

If `fSegInCore` is true, the segment is already in the heap, but not locked down. (If the segment is resident, no memory needs to be allocated for it).

`fOSErr` contains an error number. This field is valid only when the structure is passed to an error handler.
Modifications to the `RTState` structure are ignored in all cases except when the `fJTAddr` field is altered by the user error handler.

Operations

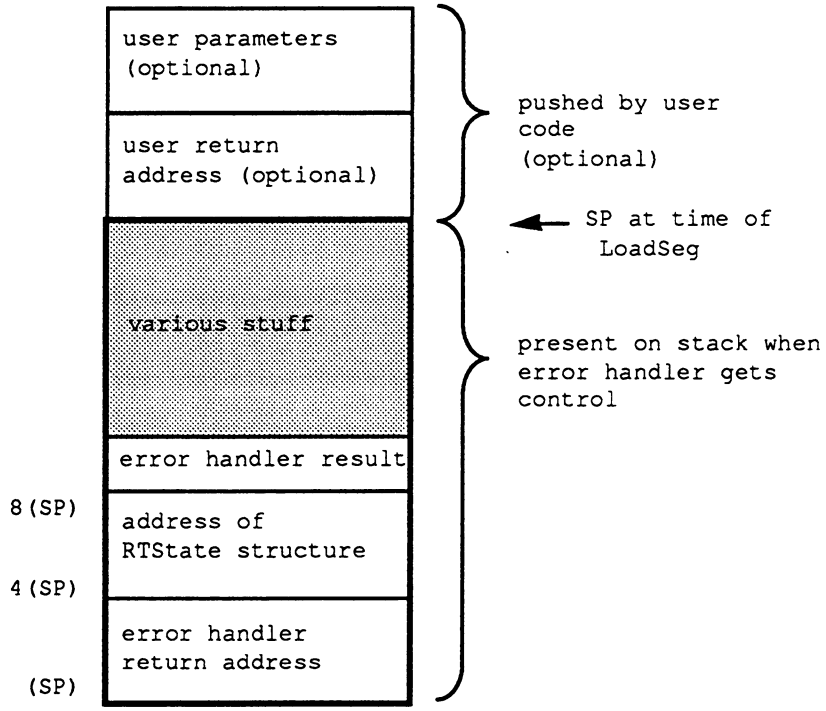
`kRTSetPreLoad, kRTSetPreLoadA5`

Arrange for the function `fUserHdlr` to be called by the segment loader just before a segment is loaded. The user's pre-load handler replaces the original pre-load handler, and a pointer to the old handler is returned in `fOldUserHdlr`. A pre-load handler could be used to ensure that enough memory is available for a segment to be loaded. When the handler returns, a segment load attempt is made. If an error occurs during the load, the error handler is invoked.

`kRTSetSegLoadErr, kRTSetSegLoadErrA5`

Arrange for the function `fUserHdlr` to be called if a segment load fails. The user's error handler replaces the old error handler, and the address of the old handler is returned in `fOldUserHdlr`.

Upon entering the error handler the stack looks like:



Items on the stack that are labelled optional may in some cases not actually appear. For instance, a simple JMP to a jump table entry would not have pushed the user parameters and return address.

The word at `8(SP)` is the space reserved for the error handler's action code.

The value at `4(SP)` points to the RTState structure which provides information about the failure.

The value at `(SP)` is the return address from the error handler. This may or may not be used depending on whether the error handler performs a `longjmp` to restore control to the application (see Retreat).

The error handler should examine the `RTState` structure and take appropriate action (e.g. release some memory, etc.). It then can exit in one of two ways:

- Take an action. Return an action code on the stack for the segment loader to act upon, and return. Current action codes are:

Value	Action
kRTRetry	Retry This restores the stack to it's original state prior to the `_LoadSeg` and re-executes the jump table entry. If all goes well, execution will continue as planned. If `fJTaddr` in `RTState` was modified, execution will resume at the new address. Note that unless preventative measures are taken (e.g. a maximum retry count) using this technique can lead to an infinite loop if retrying the load always fails.
kRTContinue	Continue This restores the stack to its original state prior to `_LoadSeg` and sets the PC to the user return address in the stack. This is dangerous since a return address may not exist.

Any other value will result in the system error, `dsLoadErr`, a segment loader error.

- Retreat! Use `longjmp` (or an equivalent) to pass control to another error handler, set up in a parent stack frame. This handler can attempt damage control (e.g. try to save the document, alert the user and quit).

`kRTSetPreUnload, kRTSetPreUnloadA5`

Arrange for the function `fUserHdlr` to be called prior to unloading a segment. The address of the original handler is returned in `fOldUserHdlr`.

The meaning of the fields within the `RTState` structure vary some from their use with the other handlers. Specifically, `fSP` points to the return address from the `UnloadSeg` call and `fJTAddr` is the address of the jump table entry used as the parameter for `UnloadSeg`. The remaining fields are the same as described above. For debuggers, the pre-unload handler can be used to uninstall breakpoints within a segment before it is unloaded.

Examples

```
{*-------------------------------------------------------------------*}
{* example.p                                                         *}
{* An example tool which installs a pre-load handler and uses it     *}
{* to print information about the segment.                           *}
{*                                                                   *}
{* pascal -model far example.p                                       *}
{* link -model far -w -t MPST -c 'MPS ' -o examplep ∂                *}
{*    example.p.o {Libraries}RTLib.o {Libraries}Interface.o ∂        *}
{*    {Libraries}Runtime.o {PLibraries}PasLib.o                      *}
{* examplep                                                          *}
{*-------------------------------------------------------------------*}

PROGRAM Example;
USES
   RTLib, ToolIntf, Types;

VAR
   p:           RTPBPtr;
   param_block:   RTPB;
   error:         OSErr;

{$S One}
PROCEDURE one;
BEGIN
   {
      do something
   }
END;

{$S Main}
```

```
FUNCTION preload_handler(state: RTStatePtr): INTEGER;
BEGIN
   {
      print segment information
   }
   WRITELN('segno = ', state^.fSegno);
   WRITELN('segtype  = ', state^.fSegType);
   WRITELN('segsize  = ', state^.fSegSize);
   IF (state^.fSegInCore) THEN WRITELN('incore = yes')
   ELSE WRITELN('incore = no');
   preload_handler := 0;
END;

BEGIN
   {
      load writeln segment so that the pre-load handler does not
      invoke another call to _LoadSeg
   }
   WRITELN('load writeln segment');

   {
      load the handler
   }
   p := @param_block;
   p^.fOperation := kRTSetPreLoad;
   p^.fUserHdlr := Ptr(@preload_handler);
   error := Runtime(p);

   {
      load the segment
   }
   one;
END.
/*-------------------------------------------------------------------*/
/* example.c                                                         */
/* An example tool which installs a pre-load handler and uses it     */
/* to print information about the segment.                           */
/*                                                                   */
/* c -model far example.c                                            */
/* link -model far -w -t MPST -c 'MPS ' -o examplec ∂                */
/*    example.c.o {Libraries}RTLib.o {Libraries}Interface.o ∂        */
/*    {Libraries}Runtime.o {CLibraries}StdCLib.o                     */
/* examplec                                                          */
/*-------------------------------------------------------------------*/

#include <stdio.h>
#include <types.h>
#include <RTLib.h>

#pragma segment One
one ()
{
   //
   // do something
   //
}
```

```
#pragma segment Main
pascal short preload_handler(RTState* state)
{
   //
   // print segment information
   //
   printf("segno     = %d\n",    state->fSegNo);
   printf("segtype   = %.4s\n",  &(state->fSegType));
   printf("segsize   = %d\n",    state->fSegSize);
   if (state->fSegInCore) printf("incore = yes\n");
   else printf("incore = no\n");
   return(0);
}

main ()
{
   RTPB  param_block, *p;
   OSErr error;

   //
   // load printf segment so that the pre-load handler does not
   //  invoke another call to _LoadSeg
   //
   printf("load printf segment\n");

   //
   // load the handler
   //
   p = &param_block;
   p->fOperation = kRTSetPreLoad;
   p->fRTParam.fSegLoadParam.fUserHdlr = (void*)&preload_handler;
   error = Runtime(p);

   //
   // load the segment
   //
   one();
}
```

Calling Launch and Chain

MPW does not provide glue or interfaces for calling `_Launch` or `_Chain`; application writers must call the traps themselves from assembly or assembly-language inlines. However, the "32-Bit Everything" environment requires some hand-holding in order to survive a call to `_Launch`.

Under 32-Bit Everything, a call to `_Launch` must be wrapped between two calls to `Runtime` using the operations `kRTPreLaunch` and `kRTPostLaunch`:

```
IMPORT      (Runtime): CODE

MOVE.W      #kRTPreLaunch,-(SP)     ; push fOperation
SUBQ.W      #2,-(SP)                ; room for result
PEA         2(SP)                   ; push ptr to RTPB
JSR         Runtime                 ; prepare for launch

_Launch                             ; attempt a launch

MOVE.W      #kRTPostLaunch,-(SP); push fOperation
SUBQ.W      #2,-(SP)                ; room for result
PEA         2(SP)                   ; push ptr to RTPB
JSR         Runtime                 ; post-launch housekeeping
```

The only parameter used from the RTPB structure is fOperation. The pre- and post-launch operations do not require a parameter block for fRTParams.

Note that DTS says that you should *never* call the _Chain trap, since it is not implemented by MultiFinder. However, if you find it necessary to call _Chain, then wrap it in the same manner as that needed by _Launch.

MPW 3.2 "411" Help
Release Notes

About "411"

"411" provides a way for Macintosh developers to achieve rapid retrieval of software development information while using Apple's MPW development system. The access can be via menus and command keys or from command line entries. The software development information includes language-specific Inside Macintosh documentation, Tech Notes, MPW command descriptions and Resource information. In addition a facility for automatic insertion of Toolbox call templates is provided. Large cross reference index files (`.WIndex`) may be optionally used to provide an extremely rapid search of the documentation for any desired word. In the absence of these files, the same search may be made, but it will be a linear search and therefore will be relatively slow.

"411" can also be customized and extended and new information can be added. The help files of "411" may be either local or on a shared file server.

Setting up "411"

"411" consists of a installation instructions file (`Read Me First`), a special UserStartup script (`UserStartup•Help`), an installation script (`Install411`), and a set of help files along with their `.index` and `.WIndex` files. It makes use of a new MPW tool, `Get`, which was written to support "411" but can be used independently. The "411" folder holds the help files, their index and cross reference index files and a "Tools" folder.

An important decision to make in setting up "411" is whether to place the "411" Help files on a server or on your local hard disk. Since these files are large (over 16 Meg total) some thought should go into deciding which files to use and whether to transfer them to your hard disk or, if you are connected to a network, to a file server. The most obvious candidate for removal is either CIncludesHelp or PInterfacesHelp. If you are not developing in both C and Pascal, one of them will probably not be needed. Less obvious, but more significant candidates for removal are the `.WIndex` files. These files are not required, but significantly speed up the cross reference search that is done when selecting the "Search" menu item. Placing the help files on a local hard disk will provide better access speed but will use significant disk space. If you have access to a file server and several persons want to access "411" Help, it may be best to move the "411" folder to the file server.

Set up "411" by writing the following two commands to your MPW WorkSheet and executing them:

```
<rls>:Install411 <info>
Execute "{ShellDirectory}"UserStartup•Help
```

where `<rls>` denotes the path to the "411" files on the release medium and `<info>` denotes the volume on which the user wishes the "411" Help files to reside. If `<info>` is omitted, the installation will be to the volume that begins the path `<rls>`. (In this latter case, the data files are not duplicated because they are already residing in the desired place.)

For example, if "411" were to be released in a folder named `411Stuff` on a CD named `MPW 3.2 Release`, and the user wanted "411" to be installed on a volume named `HelpMe`, then the commands to be executed would read:

```
'MPW 3.2 Release:411Stuff:Install411' HelpMe:
Execute "{ShellDirectory}"UserStartup•Help
```

The effect of the first of the above commands is to create the folder `HelpMe:411:` and to copy to it all of the "411" files. It then, additionally, copies the new `Get` tool to the user's MPW Tools folder, copies `UserStartup•Help` to the MPW folder, and creates a folder called `Help Folder` in the MPW folder. This latter folder contains at this time a file called `Help_Folder` whose contents is the single line: `HelpMe:411:`, i.e. the name of the folder containing the "411" information. The effect of the second of the above commands is to add the "411" menu to the menu bar, and to add one more file to the `Help Folder`, a file called `Help_Files` which contains the names of all of the "411" data files in the order in which they will be interrogated, e.g.:

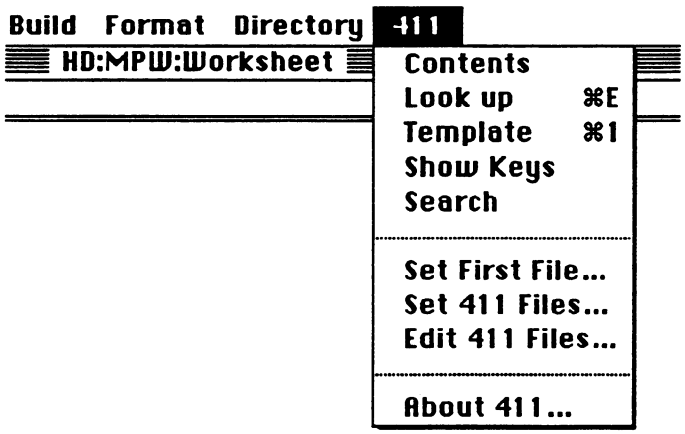
```
HelpMe:411:CIncludesHelp
HelpMe:411:InsideMacintoshHelp
HelpMe:411:MPWHelp
HelpMe:411:PInterfacesHelp
HelpMe:411:ResourcesHelp
HelpMe:411:TechNotesHelp
```

◆ Note: Because of their large size, the cross reference files (`.WIndex`) are not installed automatically. If you wish to use these files, drag their icons to the desired folder.

Using the "411" Help menu

"411" works only from within the MPW development environment. When "411" is properly set up, there should be a **411** menu on the MPW menu bar. If there has been no change to the `UserStartup•Help` script, the Help menu looks like this:

- 411 Menu



The menu items **Look up** and **Template** search all of the files listed in `Help_Files`; the items **Contents**, **Show Keys**, and **Search** look only at the first file in the `Help_Files` list. This first file is known as the *currently selected* file. It can be changed by using the **Set First File** menu item.

Contents

This menu item lets you see a list of the Help file's table of contents. For example, if the CIncludesHelp file is the currently selected Help file, then selecting the "Contents" menu item, causes a list of the ToolBox managers to appear in the Help window.

```
HelpMe:411:CIncludesHelp                Look up… "Help"
```

```
Appletalk.h       FixMath.h       Palettes.h      Serial.h
Controls.h        Fonts.h         Perf.h          ShutDown.h
CursorCtl.h       Globals         Picker.h        Slots.h
Desk.h            Graf3D.h        Printing.h      Sound.h
Deskbus.h         HyperXCmd.h     Quickdraw.h     Start.h
...
```

Note that you can obtain the same information by selecting the key word "Help" (or the name of the help file, e.g. CIncludesHelp) and then selecting the **Look up** menu item.

Look up (⌘E)

This menu item lets you look up information stored in the help files; the search starts with the currently selected file (See **Set First File...**). For example, if you choose (see 2 below) the word `FindWindow` and then select the **Look up** menu item (or type ⌘E) the following information will appear in the Help window:

```
------------------------------------------------------------------------
HelpMe:411:CIncludesHelp      Look up… "findwindow"
------------------------------------------------------------------------
short findwindow(Point *thePoint,WindowPtr *theWindow);
Type: Function
File {CIncludes}Windows.h
Trap Number A92C
InsideMacintosh Reference: FindWindow function I-287, P-35, 114, 170
FindWindow procedure  V-208
[Macintosh Plus, Macintosh SE, Macintosh II]

When a mouse-down event occurs, the application should call FindWindow
with thePt equal to the point where the mouse button was pressed (in
global coordinates, as
...
```

Thus, to get help for a given key word:

1) Choose the Help file you want information from by using the **Set First File...** menu item to make the desired file the first file (currently selected file) in the help file list. (Skip this step if the help file is already selected, or if the order of search does not matter.)

2) Click on a word in the active window. You may want to type in the word you want to look up instead. If the item is just one word, it will be automatically selected if the insertion point is adjacent to or within the word. Only if a multiple word item is to be looked up is it necessary to do a manual selection of the entire item. (Note: the means you don't have to double-click. Also you don't have to type the entire word, just enough letters to allow 411 to distinguish between the word you want and any other in the current Help file.)

3) Select the menu item **Look up** or type ⌘E. This triggers a search through the help files, in the order in which they are listed, looking for the selected key word. If the search is successful then a window named `Help` (a file in the MPW directory) is opened and the information associated with the key word is displayed, along with an indication of the file in which the key word was found.

Remember, the **Look up** menu item simply looks for the current selection in the active window.

Since numbers are keywords only in TechNotesHelp, selecting a number will retrieve the Macintosh Technical note of that number.

The header, which is placed above the "Contents" information, shows the help file that was used. To the right of the file name is a message indicating the key word on which the search was made. A mark is set to this (selected) key word in the file `Help` to aid the user in subsequent scanning of `Help` for previously gathered information.

A slight modification of `UserStartup•Help` causes the header to list all the "411" files in the order in which they are searched, with the words on the right (`Look up...`) printed on the line bearing the name of the file in which the item was actually found. (See Customizing "411" below.)

Template (⌘1)

This menu item lets you replace a toolbox function call such as FindWindow with the template for that function. For example if you were to select "FindWindow" and choose the 'Template' menu item (or type ⌘1), then your "FindWindow" selection in the Active window would be replaced by:

```
short myVariable = findwindow((Point *)thePoint,(WindowPtr *)theWindow);
```

Both C and Pascal templates are available. Use **Set First File...** on the 411 menu to choose a language by selecting either CIncludesHelp or PInterfacesHelp.

Show keys...

This menu item lets you list all of the keys in the currently selected help file which begin with the word you have selected in the Request dialog. For example, selecting this menu and then typing the two letters "fs" when CIncludesHelp is your current (first listed) help file, produces a list of all of the HFS calls that begin with "FS", i.e. FSClose, ...

Search...

This menu item lets you search the currently selected help file for all occurrences of the word you have selected. The result is a list of names (keys) whose data records contain the word. On a large Help file, e.g. CIncludesHelp, this can take a minute or more. Only the current (first listed) help file is searched even if no data record containing the word is found. If your help folder contains a cross reference file (`.WIndex`) for the Help file, the time to search is reduced to just a few seconds.

Set First File...

This menu item lets you choose a help file to be the currently selected file. A dialog window shows a list of all help files, and you are invited to make a selection. The selected file then becomes the first file in the list that appears in the file `Help_Files`. It is then known as the currently selected file, and, until you again reorder the list, is the initial target of all **Look up**, **Template**, and **Contents** requests, and the only file used by **Show Keys** and **Search**. The `Help` window opens, displaying the contents list of the selected file.

Set 411 Files...

This menu item presents a standard file dialog from which to locate a help folder. If a help folder is selected then all the files in the folder that end in the word `help` are placed in the list of files to search (in `Help_Files`). Note that this will remove any existing files from the list..

Edit 411 Files

This menu item open the `Help_Files` window, which contains a list of all the help files that "411" knows about. The list may be edited and then saved. Note that each line specifies a path to a single help file.

About 411...

This menu item displays the credits and then does a lookup in the MPWHelp file for the key "About 411".

Customizing "411"

There are several ways in which "411" can be customized by modifying the `UserStartup•Help` script:

- If you are using a file server and want the script to call the MPW Choose tool to mount the file server when MPW is launched, set the script variables `Help_Server` and `Guest`. Set the former to the desired zone:server:volume pathname, and set the latter to 1 if want to log onto the server as an AppleShare "guest.". Note: This requires the Choose tool from MPW 3.2 or later.
- If you wish to add or change command keys in the menu, simply edit the AddMenu commands in the `UserStartup•Help` script. For example, the line

  ```
  AddMenu 411 "Look up/1"
  ```

  in the script could be changed to:

  ```
  AddMenu 411 "Look up/7"
  ```

  to change the function key to ⌘7. Alternatively,

  ```
  AddMenu 411 "Look up"
  ```

  removes the function key associated with the **Look up** menu item entirely.
- If you wish to change the name of the "411" menu, modify the argument of the `AddMenu` command. For example, you can change the menu name "411" to "MyMenu" by changing all occurrences of `AddMenu 411...` to `AddMenu MyMenu...` in the `UserStartup•Help` script.

- If you wish all help files to be listed in the header, change the script so that it sets the value of the variable `headerStyle` to `-h` (the default is `-h2`).

Using the Get Tool

The retrieval of information through the "411" Help menu is based upon calls to the MPW `Get` tool. The calls that are used by "411" can be seen in the file `UserStartup•Help`. If you choose, you may instead call the `Get` tool directly or from your own script.

(See the section on Get in the MPW 3.2 Tools/Scripts Release Notes.)

Adding your own help to "411"

Help files used by the `Get` tool are ordinary MPW Shell document files whose names end with "Help" and that have an internal organization which is recognized by the Get tool. The requirements are that a Help file consists of a set of *records*, each record in turn consisting of one or more *fields*. Each record must start with the *field tag* `æKY` ("æ" is option-') followed by one or more words separated by carriage returns. The search of the help file is made on the key words. All other fields are various categories of information to be retrieved. Field tags must be the first item on a line, and are separated from the following material by one or more spaces. Field tags are case sensitive. Each field is terminated by the appearance of a new field tag. The record is terminated by the next `æKY` tag (or end of file).

Example:

```
æKY Key1
Key2
Key3
æC This is a comment
```

The various tags other than æKY are used to put the information to be retrieved into categories. The most neutral of them is æC, which is used for general textual matter. These tags in some cases modify the behavior of `Get`. For example, the tag æDT, which is used for templates, precedes data which will be retrieved if and only if `Get` is called with the `-t` option. This is used in the implementation of the "template" menu item. Other tags cause some boiler plate to be emitted prior to the text in the data base. For example, the tag æRI is used for fields that contain chapter and page references to Inside Macintosh. The text following the field tag will have inserted before it the cosmetic text: "`InsideMacintosh Reference:`".

Field Tag Codes:

æKY	Key word or set of key words separated by carriage returns. This field denotes the beginning of a "411" record and the words in this field are the record's names, i.e. the words used as keys for retrieval of the record's data.
æKL	Key word List . This is typically used in conjunction with the key "Help" to list, as a table of contents, all of the key words in the file.
æFa	File name of Assembler include file.
æFc	File name of C header file.
æFp	File name of Pascal Interface file.
æF	Used for the names of files which are not interfaces.
æT	Type of the item: function/structure/constant/etc.
æD	Formal declaration of the item: function, procedure, or structure.
æDT	A template for calls of procedures and functions.
æC	Commentary. This is general textual information, generally lengthy compared to that associated with the other tags.
æR	Reference to... Used for references other than to Tech Notes and Inside Macintosh.
æRI	Reference to Inside Macintosh. This is usually a chapter and page reference.
æRT	Reference to Tech Note. This is usually a reference by number.
æTN	Trap Number. This is used to annotate function that are in-line trap calls.
æMM	Routine may move or purge memory. This tag cause issuance of the preceding warning.

When the `Get` tool is executed, it first retrieves the byte offset of a key word from the index file, positions to the æKY line in the help file, and then reads all of the following lines until another æKY typed line is encountered. `Get` then outputs the record, after first removing the field tag codes from each field. Therefore, if the contents of a help file are altered in any way, it is necessary that the index be rebuilt. This is handled automatically by the `Get` tool, which puts up a dialog stating that the index needs to be rebuilt and requesting permission. In ordinary circumstances, this dialog should be answered affirmatively.

About the files in your "411" folder

The "411" folder contains the following files:

```
:411:
            Install411                  # the "411 installation
                                        # script.
            CIncludesHelp               # the CIncludesHelp data file.
            CIncludesHelp.index         # the CIncludesHelp index file.
            CIncludesHelp.windex        # the CIncludesHelp cross
                                        # reference index file
            InsideMacintoshHelp         # Vols. 1-7 data file.
            InsideMacintoshHelp.index   # Vols. 1-7 index file.
            InsideMacintoshHelp.windex  # Vols. 1-7 cross reference
                                        # index file.
            MPWHelp                     # the MPWHelp data file.
            MPWHelp.index               # the MPWHelp index file.
            MPWHelp.windex              # the MPWHelp cross reference
                                        # index file.
            PInterfacesHelp             # the PInterfacesHelp data
                                        # file.
            PInterfacesHelp.index       # the PInterfacesHelp index
                                        # file.
            PInterfacesHelp.windex      # the PInterfacesHelp cross
                                        # reference index file.
            ResourcesHelp               # the ResourcesHelp data file.
            ResourcesHelp.index         # the ResourcesHelp index
                                        # file.
            ResourcesHelp.windex        # the ResourcesHelp cross
                                        # reference index file.
            TechNotesHelp               # the TechNotesHelp data file.
            TechNotesHelp.index         # the TechNotesHelp index
                                        # file.
```

```
            TechNotesHelp.windex        # the TechNotesHelp cross
                                        # reference index file.
:411:Tools:
            Get                         # the Get MPW tool - used to
                                        # look up help info.
            UserStartup•Help            # the "411" UserStartup script.
```

Files created by "411"

The following files are created by the `UserStartup•Help` script, either at startup time, or as the result of execution of menu items created by the script. They all reside in the folder `"{MPW}Help Folder:"`

```
            Help                        # your "411" Help window.
            Help_Folder                 # contains the "411" path name
                                        # (path name to help data and
                                        # index files)
            Help_Files                  # contains the list of all help
                                        # files (full file names including
                                        # path)
            Help_Temp                   # temporary file used by the "411"
                                        # menu
```

MPW 3.2 Appendix A
Release Notes

StreamEdit—scriptable text editor

Syntax

StreamEdit [**-e** *string*]
[**-d**]
[**-o** *file*]
[**-s** *scriptFile*]
[**-set** *variable*[=*value*]]
[*file* ...]

Description

StreamEdit is a non-interactive text editor similar in function to the Unix® tool *sed*.[1] Providing scriptable text matching and editing operations, it is useful for making repetitive changes to files, for extracting information from text files, or as a filter.

StreamEdit takes a script and a set of input files (or standard input, if no input files are specified) and, to each line of input in turn, applies each statement in the script, writing the output to standard output or the specified output file.

A statement consists of an address or address range followed by one or more commands. The commands in a statement apply to those input lines that match the address or address range. Addresses are specified either numerically, by context matching using regular expressions, or by simple boolean functions of the above. A command may have parameters in the form of options and text strings. The commands, in general, cause modification of the input line, and may furthermore cause text to be inserted before or appended after the input line.

The **-d** option profoundly affects the editing process in that it filters out the initial input lines so that they are not sent to the output except when a particular command explicitly causes such transmission.

The script is specified in the command line by one or more **-e** or **-s** options. (Using the **-s** option, the script comes from the named file; using the **-e** option, it is the string argument to the option.) If more than one script is specified, the resulting script is the concatenation of all the scripts. If no script is specified and **-d** is not used, the action is simply to copy the input lines to the output.

A script consists of a series of statements of the form:

```
address-expression command [ ; command ... ]
```

In a script file, statements and commands are separated by newlines or semi-colons. If the script is a text string (**-e** option), only the semi-colon may be used. All the commands following a particular address expression are executed when the address of the line being processed matches the address expression.

---

[1] It is *not* compatible with Unix® `sed` or `awk`.

Note that strings must be quoted. Therefore, a command such as

```
StreamEdit -s -d script1
```

where the contents of script1 is

```
1 print 'I found line 1'; 2 print 'I found line 2'
```

should be written, using the **-e** option, as

```
StreamEdit -d -e "1 print 'I found line 1'; 2 print 'I found line 2'"
```

A command takes the form:

```
command-name [ text-arguments … ]
```

Address expressions may span multiple lines; arguments to commands are terminated by either newlines or semi-colons.

Address expressions, commands and command arguments are described below.

Empty statements and commands are legal and are ignored. However, the first command following an address expression may not be empty. Comments begin with a sharp sign (#). Semi-colons, unless they appear as the first character on a line, are equivalent to line breaks as in the Shell, and are used to terminate commands. Newlines (outside of strings) may be escaped to extend an argument list.

If a script line contains a semi-colon in the first column, the entire line is treated as a comment by StreamEdit. This allows writing StreamEdit scripts that also contain MPW shell commands. See the Examples section for more details.

Operation

StreamEdit operates on three buffers, called the edit buffer, the insert buffer and the append buffer. For each line of text in the input, StreamEdit performs the following steps:

[1] Read the next input line into the edit buffer. Clear the insert and append buffers.

[2] Evaluate each address expression specified in the script (in the order in which it appears) with respect to the edit buffer's current contents. If an address expression matches, then execute the actions associated with the expression.
The **append** command sends its argument to the append buffer; the **insert** sends its argument to the insert buffer. All other commands either affect the edit buffer or directly write to the output file.

[3] When all address/action pairs in the script have been evaluated, concatenate the insert, edit and append buffers and write them to the output file.

Note that some commands (e.g. Replace and Change) can alter the contents of the edit buffer; these changes persist, possibly affecting subsequent address matches.

Addresses

An address takes one of the following forms (operators are listed in decreasing precedence):

`( address )`	Parenthesis are used to control precedence;
`! address`	NOT operator; the address must *not* match;
`address && address`	Both addresses must match; if the first address fails, the second address is not examined;
`address || address`	Either address may match; if the first address succeeds, the second address is not examined;
`address , address`	Matches the inclusive range of lines begun when the first address matches, and ended when the second address matches (this is meant very literally; see below);
`/regular expression/`	Matches any input line containing the expression (see the chapter on Advanced Editing in the MPW Shell documentation for details; see below for extensions to regular expressions);
•	Considers a match to have succeeded *before* the first input line is read;
N	Matches input line number *N*;
$	Matches the last input line;
∞	Considers a match to have succeeded *after* the last input line is read;

Parenthesis and the `!`, `&&`, `||` and `,` (comma) operators may be used to form complex address expressions. For example:

```
(/trillian/ || /zaphod/ && /beeblebrox/) && !42
```

will match any line containing either the text "trillian", or both "zaphod" and "beeblebrox" (in any order), as long as the line is not the 42nd input line. And:

```
(1,10) && /Copyright/
```

matches a line containing the word Copyright, but only on one of the first ten input lines.

The range operator can be tricky in several circumstances. This is because it represents a two-state system, not a continuous test for a condition. When the condition for the start of the range is met, source lines are considered to match up to and including the line at which the condition for the end of the range is met. If the first of these is not met, no lines in the range will be considered. If the last of these is not met, lines will be considered to be a match forever! For example, the expression:

```
/Copyright/ && (1,10)
```

will match the range "`/Copyright/ && 1`" to "`/Copyright/ && 10`"; if Copyright does not appear on the tenth input line, the range will stay active until the end of the input, which is probably not what was intended. If Copyright does not appear on the first input line, the range will not be activated at all.

Extensions to Regular Expressions

Regular expressions are the same as those that the MPW shell supports, with the following extensions:

- A regular expression of zero length, specified with two adjacent slashes (//) means "the last regular expression matched."

  Note that the last regular expression *matched* may be different from the most recent regular expression that appears in the script. That is, the short-circuit evaluation of the **&&** and **||** operators may change the meaning of //. For instance:

  ```
  /1/ || /2/ replace // "3" -c ∞
  ```

  will replace all occurrences of "1" or "2" with "3"

- A "ç" symbol (generated by Option-C) as the first character in a regular expression causes the regular expression to match in a case-sensitive manner (normally matches are case insensitive). To use a bullet character (•) to anchor the match at the beginning of the line, the bullet must follow the "ç" symbol. That is, use:

  ```
  /ç•foo/
  ```

  and not:

  ```
  /•çfoo/              # <-- Wrong!
  ```

- The value of a variable may be referenced within a regular expression by enclosing the variable name between ≤ and ≥ symbols (this substitution is very similar to the way curly-braces work in MPW Shell scripts). The variable's value is *not* treated as a regular expression, but rather just a string that must exactly match. An empty variable (a null string) matches nothing — note that variables are empty unless they are set.

  For instance, the following script searches for all occurrences of the text "foo":

  ```
  •   set VAR "foo"
  /≤VAR≥/ print
  ```

  If the variable match has to be case-sensitive, the regular expression must be case-sensitive, as in:

  ```
  •   set VAR "foo"
  /ç≤VAR≥/ print
  ```

Common Pitfalls

By far the most common mistake for a beginning StreamEdit user to make is to confuse variables and strings in replacement expressions. When not within regular expressions, strings must be quoted; an unquoted variable name represents the value of the variable. For instance, the statements:

```
replace /hum/ dum                    # the variable dum
```

and

```
replace /hum/ "dum"                  # the string "dum"
```

differ in that the first statement replaces /hum/ with the contents of the variable dum, while the second statement replaces /hum/ with the string "dum". Since variables start out empty, if the variable dum has not been initialized then the first statement has the possibly alarming action of deleting the /hum/.

Text Arguments

Most commands accept text arguments, referred to in the command summary below as "*text*". These arguments are expanded and concatenated into a temporary output buffer and then moved to the destination. Text arguments take the following forms:

Text Argument Form	Expansion
"string" 'string'	The string itself (escape characters are processed, and single or double quotes are allowed).
.	A period represents the contents of the edit buffer (the current input line) minus its newline;
variable	The contents of the variable;
®*N*	The marked expression *N*, from the most recent match;
-from *filename*	The next line of input from the specified file, minus its trailing newline (if any). The filename argument may be any other text argument form except **-n** (a string, a variable, an ®-variable, etc.). If the filename is empty (e.g. an uninitialized variable) the value is empty.
-n	Suppress trailing newline at end of expansion.

Variables are names for strings. Variables are initially empty. Variable names are case-ignored C identifiers, that is, [a-z_][a-z0-9_]*. Again, care should be taken not to mistake variables for strings.

Numbers are decimal. Strings are enclosed by single or double quotes. The backslash (\) and shell-quote (∂) characters may be used to quote special characters within strings and regular expressions:

Quote form		Expands to
∂n	\n	newline
∂t	\t	tab
∂\	\\	backslash
∂∂	\∂	shell quote

Files are named streams of text; the argument to **-from** or **-to** may be a string, a variable, an "®" variable, or even a **-from**. It is possible to read a file that is also being written; writing to a file does not change the read position. There can be any number of files; StreamEdit is not limited by the system's FCB count.

Commands

The underlined portions of each command name below indicate the minimum amount of text needed to specify the command. It is guaranteed that future commands will not conflict with existing abbreviations.

Append [-**n**] *text*

Append the specified text to the append buffer, to be written at the end of the current cycle. If -**n** is not specified, a newline is automatically added to the end of the text.

```
$ Append "This line follows the last line of the input"
42 App -n "This is line 43 ==>"
```

Insert [-**n**] *text*

Append the specified text to the insert buffer, to be written at the end of the current cycle. If -**n** is not specified, a newline is automatically added to the end of the text.

```
1 Insert "This text precedes the first line of the file"
42 Ins "This is line 42 ==>" -n
```

Change [-**n**] *text*

Replace the contents of the edit buffer with the specified text. If -**n** is not specified, a newline is automatically added to the end of the text.

```
42 Change "This is line 42"
/droid/ Ch "These are not the droids you're looking for"
```

Delete

Clears the contents of the edit buffer; essentially the same as specifying "`Change -n`".

```
1,$ Delete
/•[ ∂t]*#/ Delete     # delete lines that are Shell comments
```

Next

Concatenates and prints the insert, edit and append buffers. Gets the next line of input and begins the match-execute-print cycle over again.

```
1,/foo/ Next          # ignore input until "foo" is found
```

Note that **Next** affects the normal control flow in the script, sometimes with unexpected results. Once the **Next** command has been executed, no further commands in the script that match the current line will be executed, because the next line of input is fetched immediately. Note also that the printing is independent of the **-d** option; the buffers will be printed exactly **once** regardless of the presence or absence of **-d**.

Print [**-appendto** *file* | **-to** *file*] *text*

Print the specified text on standard output, or to file specified by **-to** or **-appendto**. The text is written immediately (that is, before the insert, edit and append buffers are written). If no text is specified, ".", the contents of the edit buffer, is assumed. If -**n** is not specified, a newline is automatically added to the end of the text.

The **-appendto** option appends the output of Print to an existing file. If the file does not already exist, it is created.

The **-to** option directs the output of Print to the specified file. The file is truncated the first time it is written to.

Both -**append** and -**to** require a filename argument. The filename may be specified by a string, a variable, or an ®-variable. If the filename is an empty string (e.g. an uninitialized variable) nothing is printed.

```
1,10 Print
1,$ print -to "MyFile" ">>> " .
/([a-z0-9]+)®1/ print -to ®1
1,$ Pr -to MYVAR
```

Replace [-**c** *count*] */pattern/ replacement_text*

Replace the pattern in the edit buffer with the specified text. Unlike their behavior in other commands, any "®" variables in the replacement text refer to the values set by processing the *pattern* argument to Replace, not the "®" variables in the line's address.

The count following -**c** may be a number, or "∞" (which specifies an infinite count).

```
/42/ replace /42/ "Meaning of Life"
/42/ replace // "Meaning of Life"          # // is the same
Rep /(©[ ∂t]*19[-0-9]+)®1([ ∂t]*Apple≈)®2/ ®1 ",1989" ®2
```

Exit [*status*]

Stop processing. Nothing more is printed. If a single numeric argument is supplied, it is used as StreamEdit's exit status.

```
Exit 42              # exit with status 42
Exit                 # exit with status 0
```

Set [-**a** | -**i**] *variable text*

Set the contents of the variable to the specified text. No newline is automatically added. The -**a** option causes the text to be appended to the variable's current contents. Likewise, the -**i** option causes the text to be inserted before the variable's current contents.

```
/•#!([a-zA-Z0-9_]+)®1/ Set current_file ®1
set line .           # make copy of current line, sans newline
set line . "\n"      # make copy of current line, with newline
set -a frog -from "some file"
```

Option *keyword* ...

A general purpose command that controls the processing of the script and the input text. Current keywords are:

AutoDelete

Specifying the "auto delete" option is equivalent to appending the command

```
/≈/ Delete
```

to the very end of the script. All input lines will be deleted; the only output will be from **Print** commands. This is also equivalent to the -**d** command line option. This is useful when writing filters or scripts that do not need to copy the input lines as a matter of course.

Type Tool

Input Standard input, if no files are specified.

Outputs The result of applying the script to each line of input, directed to standard output unless the **-o** option is specified.

Also, the results of executing **-to** *file* and **-appendTo** *file* commands.

Options

-e *string* The string is the script to compile. This option may be used more than once, and may be used in conjunction with the **-s** option. The final script is the concatenation of the **-e** and **-s** options, in the order specified on the command line. Note that *string* must be quoted.

-d Specifying this is exactly equivalent to having a command of the form

```
/≈/ Delete
```

as the very last command in the script. It causes all input lines to be deleted; the only output will be from **Print** or **Next** commands. This is also the same as specifying the **"Option AutoDelete"** command (see above).

-o *file* Direct the final output of StreamEdit to the specified file. **-o** is a "safe" option; the destination file is not written until all of the input is read, so the output file may be one of the input files.

-s *file* Read the specified file and compile the script it contains. This option may be used more than once, and may be used in conjunction with the **-e** option. The final script is the concatenation of the **-e** and **-s** options, in the order specified on the command line.

-set *variable*[=*value*]
Set the variable to the specified value; this is exactly like using a **Set** command, except that the variables are defined before the script is executed. The value may be omitted, in which case the variable will be set to the empty string.

Examples

Extracting the Leaf Part of a File Name

It is sometimes necessary to extract the leaf part of a complete file path name in a Shell script. The StreamEdit expression:

```
/(≈:)*([¬:]*)®1/
```

sets the variable ®1 to the part of the file name following the last colon, or to the whole file name it doesn't contain a colon. It could be used in a Shell script as a filter:

```
# {MPW}Scripts:FilterLeaf
StreamEdit -d -e '/(≈:)*([¬:]*)®1/ print ®1'
```

For example:

```
Echo "The:I:Is:Silent:myFile" | FilterLeaf
```

would print:

```
myFile
```

Generating Inlines

This is a script that generates MPW C or C++ inline function declarations from assembly language source. It is far easier and less error-prone than hand-assembly or cut-and-paste; even though this script depends on the format of the listing file produced by the MPW Assembler, it is better to automate the process.

The script's usage is:

```
MakeCInline assemblyfile.a >outputFile
```

Given assembler input something like this:

```
;+
;  Inline Pascal string copy
;
;¥ void pascal_string_copy(char* src, char* dest);
;
;-
            proc
            movem.l     (SP),A0-A1
            moveq       #0,D0
            move.b      (A1),D0
            bra.s       @2
@loop:      move.b      (A1)+,(A0)+
@2:         dbra        D0,@loop
            endproc
```

We want the filter to produce an inline declaration something like this:

```
void pascal_string_copy(char* src, char* dest) =
{0x4cd7, 0x0300, 0x7000, 0x1011, 0x6002, 0x10d9, 0x51c8, 0xfffc};
```

The character "¥" in the assembler comment marks the declaration; in principle any unique character or string can be used to flag the declaration.

The script has two parts; the first part contains MPW Shell commands, the rest of the script contains StreamEdit statements.

The MPW Shell part of the script is:

```
# MakeCInline -- make C assembly language inline declarations
; asm "{1}" -l
; StreamEdit -d -s "`which {0}`" "{1}".lst
; Delete "{1}".lst "{1}".o
; exit
```

It runs the assembler on the input file, producing a listing which is processed by the rest of the script. Then the temporary files are removed and the MPW Shell part of the script exits; the Exit command ensures that the Shell doesn't execute any StreamEdit statements.

The invocation of StreamEdit here uses an interesting trick; the name of the StreamEdit script to execute is, naturally, the name of the currently executing script. So we use

```
`which {0}`
```

which expands into the name of the currently running shell script.

The rest of the file contains a StreamEdit script that processes the assembly listing produced above. Here is an example of the assembler's listing output:

```
MC680xx Assembler - Ver 3.2d1          21-Nov-89  Page   1
Copyright Apple Computer, Inc. 1984-1989

Loc   F Object Code     Addr  MSource Statement

                                          case       on
                        ;+
                        ;
                        ; ¥ void pascal_string_copy(char* src, char* dest);
                        ;
                        ;-
00000                             strcpy     proc       export
00000   4CD7 0300                            movem.l    (SP),A0-A1
00004   7000                                 moveq      #0,D0
00006   1011                                 move.b     (A1),D0
00008   6002          0000C                  bra.s      @1
0000A   10D9                      @loop:     move.b     (A1)+,(A0)+
0000C G 51C8 FFFC     0000A       @1:        dbra       D0,@loop
00010                                        endproc

                                             end

Elapsed time: 0.08 seconds.

Assembly complete - no errors found.  16 lines.
```

The opcodes we need are tantalizingly close, but embedded in material that we need to strip away. The first job is to extract the inline's declaration and copy it to the output. Hex constants must be separated by commas—we accomplish this with a variable, initially empty, that is set to a comma-and-space when a hex constant is emitted, so that a comma precedes every hex constant but the first one.

```
/;¥[ ∂t]*([¬;]*)®1/
        Print -n ®1 " =∂n{"
        Set PRECEEDING_COMMA ""
        Delete
```

The regular expression matches the inline declaration in the comment (which can be recognized by virtue of the marker string, "¥", that we put there). The text of the inline is extracted, omitting a possible trailing semicolon, and put into the variable ®1. The Print statement emits the inline declaration (in ®1) and extra stuff needed for C inline syntax. The PRECEEDING_COMMA variable is set to empty, the line is deleted, and processing continues.

The inline declaration is terminated by an ENDP or an ENDPROC assembler directive:

```
/[ ∂t]ENDP/
        Print "};∂n"
        Delete
```

Next, totally uninteresting lines are deleted. Examining the assembly listing, we note that the lines with the object code we need invariably contain a hex constant starting in the

first column, several spaces (with an optional "G"), and at least one two-byte hex constant. We'll strip every line that doesn't meet these criteria, so that there will be less noise to worry about.

```
!/•[0-9a-f]+ [ g] [0-9a-f]«4»/
        Delete
```

Then we simply delete any junk that precedes the hex constant we're interested in:

```
1,$     Replace /[0-9a-f]+ [g ] /  ""
```

Now the line contains one word of assembler output that we can copy to the output:

```
/•([0-9a-f]«4»)®1 /
        Print -n PRECEEDING_COMMA "0x"®1
        Set PRECEEDING_COMMA ", "
        Replace // ""
```

We print an optional comma, followed by the hex constant itself. Then we arrange for future constants to be preceded by a comma, and remove the constant from the front of the line.

Now we have a problem—there's no way to tell how many more constants have to be processed on the line under consideration. Furthermore, StreamEdit has no control structures for looping, so a count wouldn't help much anyway. We resort to an artifice, namely, repeating the above code as many times as we're likely to ever need it for a single line of assembler output.

```
#
#  Convert remaining words on line
#
/•([0-9a-f]«4»)®1 /
        Print -n ", 0x"®1
        Replace // ""
/•([0-9a-f]«4»)®1 /
        Print -n ", 0x"®1
        Replace // ""
/•([0-9a-f]«4»)®1 /
        Print -n ", 0x"®1
        Replace // ""
/•([0-9a-f]«4»)®1 /
        Print -n ", 0x"®1
        Replace // ""
```

Examination of the assembler output shows that handling five constants on a line is more than enough. However, if the assembler listing format changes, the script will break.

Unpacking Unix Shell Archives

This script unpacks a Unix shell archive, more commonly known as a *shar* file. Shar files are used in the Unix community to gather text (say, the sources for a program, including its makefile) into a single file, suitable for transmittal by electronic mail or usenet.

Shar files typically have the form:

```
garbage at the beginning – mail headers and so forth
sed "s/^X//" >TheFile <<'END_OF_TheFile '
Xtext of the file
X  where each line
X    is preceded by an 'X'
END_OF_TheFile
more files, similarly packed
```

The Unix shell and the tool sed cooperate to strip off the "X"s at the beginning of each line, and to direct the output to the correct file. Unfortunately the MPW Shell does not have this kind of redirection, and StreamEdit is not *sed*, so we have to come up with our own solution.

To make matters worse, there is no single format for a shar file—in the Unix community it's "anything goes," as long as the standard Unix tools can unpack it. A StreamEdit script to unpack an arbitrary shar file would have to closely emulate the Unix environment, which is rather difficult. In practice, you will have to tweak this script to handle different kinds of shar files.

The script starts with the usual MPW Shell commands to start up StreamEdit with the proper script, pass along the command-line parameters, and exit.

```
; streamedit -d -s `which "{0}"` {parameters}
; exit
```

The variable FILE holds the name of the current output file. When we see a line beginning with "sed", we extract the output filename (possibly enclosed in quotes) and put it in the variable.

```
•       Set FILE "DELETE.ME"                    # for safety's sake

/•sed/ &&    ( />[ ∂t]*∂'([¬ ∂t]*)®1∂' /     # sed … >'quotedFile'
               ||
               />[ ∂t]*([¬ ∂t>]*)®1 /         # sed … >notQuotedFile
             )
        Set FILE ®1
        print "Extracting " FILE
```

For paranoia's sake, the FILE variable is initialized to "DELETE.ME", and the name of each file extracted is printed on standard output.

Extraction is simple—for every line beginning with an "X", the "X" is stripped off and the line is written to the current destination file.

```
/•X/
        replace // ""
        print -to FILE
```

Files `{ShellDirectory}StrEd.≈.tmp` Temporary output file.

Limitations Lines of more than 1,000 characters are silently split.
There is no easy way to do a Replace operation on a variable.
There are no "true" expressions — arithmetic is impossible.
There are no conditionals or other control structures.

See Also Shell documentation on regular expressions.

MPW 3.2 Appendix B
Release Notes

CMarker -- C++ / ANSI C Recognizer, Preprocessor and Marker Tool

Syntax CMarker [*option...*] [*file...*]

Description CMarker reads the specified C++ / ANSI C source file(s), syntax checks them and generates appropriate "Open" and "Mark" MPW commands, which, when executed, will mark the source file(s) at each function definition with the marker name being the name of the function. It's purpose is to aid in the marking of source files for use with the MPW "marker browser " capability. CMarker contains a full ANSI C preprocessor and provides options to mark include files, generate source listings (with or without showing macro expansions), run the preprocessor only, flag anachronisms, and syntax check C++ / ANSI C with or without Apple extensions.

Type Tool.

Input If no filenames are specified, standard input is parsed. Each file specified on the command line is parsed separately.

Output "Open" and "Mark" commands are written to standard output for subsequent execution by the user. Additionally, if the **-l[ist[ing]]** option is specified, the source listing is written to standard output. Preprocessor output may optionally be written to the the file specified by the **-ppout** option.

Diagnostics Errors, warnings, and anachronisms are written to diagnostic output. If the **-p[rogress]** option is specified, progress and summary information is also written to diagnostic output.

Status The following status codes may be returned:

0	No errors.
1	Parameter or option error.
2	Execution error.
3	Syntax errors.

Options Options may appear in any order and may be interleaved with the file names. All options apply to the compilation of all the files.

-a[nachronisms] Suppress anachronisms messages. By default, warnings for obsolescent ANSI C features and C++ anachronisms are written to diagnostic output.

-d[efine] *name*[=*string*] [,*name*[=*string*]]... Define *name* to the preprocessor with the value 1 if the string is omitted, or with the value of the *string*. This is the same as writing #defines for these names at the beginning of each source file.

-e

Show macro expansion in the listing (**-l[ist[ing]]**) and/or preprocessor output files (**-ppout**). Note, this option is assumed if a preprocessor output file is specified (**-ppout**).

-errors

Suppress marking if errors are detected. The default (i.e., not specifying **-errors**) is for CMarker to generate marker commands, regardless of syntax errors. CMarker can generate erroneous marker commands if the errors are sufficiently severe to confuse the parser; specifying **-errors** will cause CMarker to terminate before emitting marker commands.

-ext[ensions] **on** | **off**

Control whether Apple extensions are supported. Extensions include the SANE data types, Pascal declarations, etc. By default, the extensions are supported (i.e., **on** is assumed). Setting the extensions **off** will result in the Apple extensions generating syntax errors. Note, with extensions on, the value of the macro `__STDC__` is set to 0.

-i[ncludes] *pathname*[,*pathname*],...

Search for include files in the specified directories. Multiple **-i[ncludes]** options may be specified. At most 14 directories are searched. The set of directories should be the same as that used for compilation in order to get all the preprocessor definitions.

The search order is:

1. The include filename is used as specified. If a full pathname is given, then no other searching is applied. If the file isn't found, and the pathname used to specify the file is a partial pathname (no colons in the name or a leading colon), then the following directories are searched.
2. The directory containing the current input file.
3. The directories specified in **-i[ncludes]** options, in the order listed.
4. The directories specified in the MPW Shell variable {CIncludes}.

-lang[uage] **C** | **"C++"**

Specify the target sources as either C or C++. The default is C. If you specify C++ explicitly, it must be quoted in MPW.

-linesize *n*

Specify the maximum number of characters generated in a single listing line before a newline is inserted to fold the line. The default, for all practical purposes is infinity, or more precisely, about 1024.

-l[ist[ing]]	Generate a listing of the source to standard output. The listing includes line number information (total line and file line). If **-showskipped** is specified, lines which are skipped in the input due to conditionals are also shown and flagged accordingly.
-mc68881	Define the macro name `mc68881` as having the value 1. This is the same as writing a #define for this name at the start of each source file. If this option is not specified, then the macro name `mc68881` remains undefined unless defined by the **-d[efine]** option above. This option is only provided for compatibility with MPW C because the macro it generated may be used in the source.
-msi	Generate mark commands for include (header) files. Specifying this option will generate mark commands for methods (function elements) defined in header (".h") files which are "#included" in the source. The default is to inhibit marking header files (Applicable to C++ files only).
-pp	Preprocessing only. The syntax parser is turned off. Mark commands will not be generated.
-ppout *filename*	Write the preprocessor output to the specified *filename*. The preprocessor output is essentially the same as the listing output, but with all additional listing information removed. As such, it could be used as a source input file to a compiler. Note, specifying the **-ppout** option implies the **-e** option to show macro expansions.
-p[rogress]	Write CMarker's version, progress, and summary information to diagnostic output.
-showskipped	Show lines skipped by conditional compilation in the listing output.
-t	Display processing time and number of lines to diagnostic output even if progress information (**-p[rogress]**) is not being displayed.
-u[ndefine] *name*[,*name*]...	Undefine the predefined preprocessor symbol *name*. This is the same as writing #undef for the *name* at the beginning of each source file. This option is provided for compatibility with MPW C. The same names predefined in MPW C are predefined here, e.g., `__FILE__`, `__LINE__`, etc. See also the **-mc68881** option above.

Limitations CMarker contains a parser that is only a C++ / ANSI C syntax recognizer. It is not a compiler and does not have any of the semantic constraints that can be detected by a compiler. Where MPW C and ANSI C differ, the grammar will only accept ANSI C. The preprocessor is also more fully ANSI C than MPW C (particularly in the way macros are handled). This means that sources containing certain "old-fashioned" C constructs will be flagged in error and in some cases confuse the parser enough to generate several errors. Generally, however, the markers will be generated correctly.

In C++ and ANSI C there are many semantic constraints imposed on the syntax. These are nonexistent here. Therefore, C++ / ANSI C source which is syntactically correct and accepted by the parser has no guarantee of successful compilation. It should be remembered, however, that the intent of CMarker is not to provide a syntax checker (although it does a pretty good job of that), but to generate "Mark" commands for the MPW Marker Browser.

Known Bugs CMarker will occasionally emit erroneous mark commands if the parser has detected errors of sufficient severity as to get confused. See the **-errors** option above.

MPW 3.2 Appendix C
Release Notes

Get -- Retrieval and Indexing Tool for Large Text Files

Syntax:

Get (*dataFile...* | -dfl *listFile*) [-x] [-k *key*] [-col *n*] [-d *default key*] [-h | -h2] [-l] [-nf] [-q] [-s] [-search] [-t] [-sfl] [-y] [-field *field list* [-format *format string*]] [-lessFields *field list*]

If `Get` is successful then the record in the data file associated with `key` is written to the standard output. If it is unsuccessful then the error message "### key NOT found" is written to standard error.

Status:

0:	no error, search was successful
1:	syntax error
2:	error in processing
3:	system or out of memory error
4:	key not found (-k key)
-9:	user abort

Options:

dataFile...: One or more specially formatted files, each of which must be accompanied by an index file whose name is of the form `dataFile.index`, and whose file type is `btre`.

-dfl listFile: This is an alternative to the dataFile parameter. `listFile` specifies a file which contains a list of data file path names.

-k key: A key word in the index file (`.index`). A search is made for this key word; the search starts with the first listed data file and continues through the datafiles in the order in which they are listed, either as the `dataFile` parameter or as the list in `listFile`. There is a side effect of the search in the special case that the key is the file name (the terminal name, not the full path) of one of the data files and the `-dfl listFile` option is used. The side effect is that the file name used as a key will be moved to the top of the list in `listFile`.

-width w: Display results in multi-column format to fit a display window whose width, in characters, is w. Must be a number between 1 and 200. This option applies only to *key lists*, either specified explicitly by the tag `æKL` in the data, or implicitly by use of the `-l` or `-search` options. The option, when applied to other kinds of results, is ignored.

-d default: If the key word parameter is null, use "default" as the key word. Example: `get MPWHelp -k "" -d help` will use "help" as the key word. `get MPWHelp -k Asm -d help` will use "Asm" as the keyword. This is useful in scripts where a possibly omitted keyword is passed as a parameter.

-h: Output a header, e.g.:

```
hd80:411:MPWHelp
hd80:411:CIncludesHelp    Look up… "FindWindow"
hd80:411:InsideMacintoshHelp
hd80:411:PInterfacesHelp
hd80:411:ResourcesHelp
hd80:411:TechNotesHelp
```

```
<<data for "FindWindow" goes here...>>
```

List all data files in the header. The "Look up" line appears opposite the file actually used.

-h2: Similar to `-h`, but include in the header only the single data file that is actually used.

-m Select the key word that was found. (This must be used in conjunction with `-h` or `-h2`, which causes a display of the key-word in the form: `Look up... keyword`.) Assign a marker to the selection and set `SaveOnClose` for the active window to `False`.

-l: List all keys that begin with the letters of the named (`-k key`) key word . If no key word is specified, then list all keys in the data file. Only the first listed data file is searched

Example: The command `get MPWHelp -k asm -l` produces the list:

```
Asm
AsmCvtIIGS
AsmIIGS
AsmMatIIGS
```

-nf: No filtering. Return the key word's data exactly as it appears in the data file (including field tags).

-q: Quiet! Don't output `### key NOT found` when a key is not found.

-s: Select the key word from the current selection in the active window. For obvious reasons, this can only be used in a script activated from a menu.

-search: Search the data file for all occurrences of key word (`-k key`) and return a list of all keys whose records contain that key word. Only the first listed data file is searched.

-t: Output only a template of the function or procedure requested.

-field field list This option specifies which of the data fields, associated with a key word, to display. It is a comma-separated list of case sensitive field tags.

Example: `-field C,KL,T,Fc`

-format format string This option may be used in combination with the `-field` option. It is used to specify string information that is to be put in front of the data associated with a given field tag. The string `%s` terminates the string information for a given tag and represents the data associated with that field tag. The set of `%s` symbols is in one-to-one correspondence with the set of field tags in the field list of the `-field` option.

Example:

`get cincludeshelp -k sin -field Fc,D` produces as output

```
{CIncludes}Math.h
extended sin(extended x);
```

```
get cincludeshelp -k sin -field Fc,D
     -format "The file is: %sThe declaration is: %s"
```

produces as output

```
The file is: Math.h
The declaration is: extended sin(extended x);
```

-sfl Produces an ordered list of the requested data files. This is useful only in conjunction with the option `-dfl listFile`.

-y The index file is automatically built if it is out of date or missing. This option causes it to happen silently, In the absence of the option, a dialog will be presented asking whether the index file should be built.

-x Build, or if it is out of date, rebuild the cross reference index file for the named help file.

Example: `Get NewHelpFile -k <anyKey> -x` will build or rebuild the cross reference index file `NewHelpFile.WIndex`.

MPW 3.2 Appendix D
Release Notes

ProcNames—display Pascal procedure and function names

ProcNames has been enhanced to have the ability of generating MPW Shell *mark* commands to place markers on all the procedures and functions in a Pascal file.

New and Modified Options:

-b	Display line number information for the the start of the procedure or function body (i.e., its BEGIN) instead of the header. If marker commands are being generated (**-m**), the markers will be placed on the procedure or function BEGIN that delimits its body.
-cond	Process Pascal $setc and $ifc, $elsec, $endc conditionals. This option is assumed if **-d**, **-MC68020**, or **-MC68881** options are specified. Pascal conditionals must be processed if you process USES statements (**-u** option) and those USES reference {PInterfaces}. Due to the way these interfaces are organized, ProcNames will not parse the Pascal source correctly unless the Pascal conditionals are processed. ProcNames will not list routine names which are skipped due to conditionals.
-d *name*=TRUE | FALSE,...	Set the compile time ($setc) variable *name* to TRUE or FALSE. This option has the same meaning and effect on compile time conditional as in the Pascal compiler. The purpose is to set initial value for variables tested by Pascal compile time conditionals statements ($ifc). The **-cond** option is assumed if **-d** is specified. Caution: the **-d** option is "overloaded"! If a Shell command line parameter of the form *name*=TRUE | FALSE is specified, then this define form for the **-d** option is assumed. Otherwise the "reset to line 1 for each now file" form is assumed. For obvious reasons, a filename of the form id=id must not be placed immediately after a **-d** option.
-l *n*	Process procedures and functions only to maximum nesting level *n*.

-m	Generate MPW Shell marker commands *instead* of a display of procedure names. The marker strings will be the corresponding procedure or function names indented as a function of their nesting level. For units, marker commands will also be generated for the INTERFACE procedure and function declarations. The markers may be placed on the procedure header names (by default) or the BEGIN statement of the procedure's body (using the **-b** option). You may want to use the **-l** option to limit the markers to only the n-most outer procedures.
-MC68020	Specify this option if the source has any Pascal directive conditional of the form {$ifc OPTION(MC68020)} and you use that fact to generate MC68020 code from the Pascal compiler. The **-cond** option is implied by using this option.
-MC68881	Specify this option if the source has any Pascal directive conditional of the form {$ifc OPTION(MC68881)} and you use that fact to generate MC68881 code from the Pascal compiler. The **-cond** option is implied by using this option.
-o	The source file is an Object Pascal Program. The identifier OBJECT is considered as a reserved word so that Object Pascal declarations may be processed. The default assumes that the source is not an Object Pascal program. If markers are being generated (**-m**), then marker commands will be generated to mark the names of all objects declared in the input file.

Limitations changes/additions:

The limitation that recommends against using the **-u** option to request processing of USES declarations is no longer applicable. The following paragraph should be added to "limitations":

> When Pascal compile time conditionals are processed, the form {$ifc OPTION(id)) is only fully supported when the id is MC68020 or MC68881. In all other case this form of $ifc evaluates to TRUE. The Pascal $MC68020 and $MC68881 are treated as special cases and tracked by ProcNames. The only reason for this is to handle the {$ifc OPTION(id)} conditional. It is unreasonable and impracticable to have this ProcNames track every Pascal compiler option that exists or may exist in the future. MC68020 and MC68881 have been singled out since they, of all the Pascal options, are the most likely candidates to be used generally in Pascal source. Most other options are very specific to the Pascal compiler itself and are unlikely to be used. Indeed, the $MC68881 is explicitly tested in SANE.p. Currently there are no other uses of the {$ifc OPTION(id)} form in any other of the standard Pascal libraries.

MPW 3.2 Appendix E
Release Notes

PasRef—Pascal cross-referencer

New options have been provided for this tool, and changes have been made to the "Limitations" section as given in MPW 3.0 Reference, Vol.2.

New options:

-cond	Process Pascal $setc and $ifc, $elsec, $endc conditionals. This option is assumed if **-d**, **-MC68020**, or **-MC68881** options are specified. Pascal conditionals must be processed if you process USES statements (**-u** option) and those USES reference {PInterfaces}. Due to the way these interfaces are organized, PasRef will not parse the Pascal source correctly unless the Pascal conditionals are processed. PasRef will not list in the cross-reference identifiers in statements which are skipped due to conditionals. In the generated source listings, conditionally skipped lines are flagged with a "°".
-d *name*=TRUE | FALSE,...	Set the compile time ($setc) variable *name* to TRUE or FALSE. This option has the same meaning and effect on compile time conditionals as in the Pascal compiler. The purpose is to set initial values for variables tested by Pascal compile-time conditional statements ($ifc). The **-cond** option is assumed if **-d** is specified. Caution: the **-d** option is "overloaded"! If a Shell command line parameter of the form *name*=TRUE | FALSE is specified, then this define form for the **–d** option is assumed. Otherwise the "reset to line 1 for each now file" form is assumed. For obvious reasons, a filename of the form id=id must not be placed immediately after a **-d** option.
-MC68020	Specify this option if the source has any Pascal directive conditional of the form {$ifc OPTION(MC68020)} and you use that fact to generate MC68020 code from the Pascal compiler. The **-cond** option is implied by using this option.
-MC68881	Specify this option if the source has any Pascal directive conditional of the form {$ifc OPTION(MC68881)} and you use that fact to generate MC68881 code from the Pascal compiler. The **-cond** option is implied by using this option.

Limitations changes/additions:

In the limitation that specifies the number of symbols Pasref can handle, the number has been changed from 5000 to 6000.

The limitation that recommends always using the **-nu** option to suppress processing of USES declarations is no longer applicable. The following paragraph should be added to "limitations":

> When Pascal compile time conditionals are processed, the form {$ifc OPTION(id)) is only fully supported when the id is MC68020 or MC68881. In all other case this form of $ifc evaluates to TRUE. The Pascal $MC68020 and $MC68881 are treated as special cases and tracked by ProcNames. The only reason for this is to handle the {$ifc OPTION(id)} conditional. It is unreasonable and impracticable to have this ProcNames track every Pascal compiler option that exists or may exist in the future. MC68020 and MC68881 have been singled out since they, of all the Pascal options, are the most likely candidates to be used generally in Pascal source. Most other options are very specific to the Pascal compiler itself and are unlikely to be used. Indeed, the $MC68881 is explicitly tested in SANE.p. Currently there are no other uses of the {$ifc OPTION(id)} form in any other of the standard Pascal libraries.

MPW 3.2 Appendix F
Release Notes

FileDiv—divide a file into several smaller files

The functionality of FileDiv has been expanded. FileDiv now allows an input file to be viewed as containing an arbitrary byte stream in its data fork. The first paragraph of "Description" (MPW 3.0 Reference, Vol. 2) should be amended to read:

> FileDiv is the inverse of the Concatenate command. It is used to break a large file into several smaller pieces. The input file is divided into smaller files, each containing an equal number of bytes or lines determined by the *splitpoint* (default=2000 lines or 10000 bytes). The last file contains whatever is left over. The file to be read can either be viewed as a sequence of TEXT file lines (the default), or as an arbitrary typed file with a byte stream in its data fork (using the **-b** option).

The following options have been changed:

-f	....add the following sentence: The **-f** option is ignored if the **-b** option is specified.
-n *splitpoint*	Depending on whether or not the **-b** option is specified, split the input file into, respectively, groups of *splitpoint* bytes or *splitpoint* lines. (Note that for the case of lines, the **-f** option causes splitting into groups of *splitpoint* or more lines.)

The following options have been added:

-b	The input file is viewed as an arbitrary typed file with a byte stream in its data fork. This file is divided into groups of bytes with each group containing up to the number of bytes specified by the *splitpoint*. If this option is omitted, the file is assumed to be a TEXT file composed of a sequence of lines.
-s *N*	When the **-b** option is used, the input/output byte streams are buffered in buffers whose size are N*512 bytes. The default value for N is 128 (yielding 65536 byte buffers). Use this option to change N. Values 1 to 512 are allowed.

MPW 3.2 Appendix G
Release Notes

PROJECTOR

An Informal Tutorial

Introduction

Projector is an integrated set of tools and scripts whose primary purpose is the control of source code. The system has two basic functions. The first is to make it practical to have several people working simultaneously on a project. By allowing only one person at a time to modify any given file, it prevents a programmer from inadvertently destroying changes made by another. The second function is to preserve, in an orderly manner, revisions of a file and commentary on the revisions. This enables programmers to find out the author and revision date, to read the revision commentary, and if desired, to retrieve the revision itself.

Projector represents a considerable advance over older systems, such as SCCS and its descendants, known to users of UNIX®. It uses its own window interface for three of its commands (NewProject, CheckIn, and CheckOut). These windows, unlike Commando windows, can stay open indefinitely. The Commando interface is also available for all commands, although its use is not recommended for the three commands just mentioned. Projector also differs from SCCS in that its use is not restricted to text files. However, the data compression achieved by storing only one complete copy of a file and storing revisions as files of differences is only available for text files. Projector also has a degree of flexibility which permits different users of the same set of files to view them differently. This is accomplished by giving each user independent control of the mapping between the local directory hierarchy into which he/she keeps the files and the hierarchy used for their storage in the Projector database. Finally, Projector has a facility for associating a specific set of file revisions with a name, this name being usable as a designator for a particular version, or release, of a product. Thus, the name alone can be used to trigger the selection of just those sources that are required to build the desired instance of the product.

This tutorial begins with a section that discusses the basic concepts and terminology. Following this are sections that demonstrate the use of Projector by creating a database skeleton from scratch, putting files into it, and performing various revision activities. These sections are illustrated with screen shots taken during the actual operations.

Concepts and Terms

The top level, fundamental construct in a Projector database is called a *project.* Projects are analogous to directories in an HFS (hierarchical file system). A project may contain files and may contain other projects; just as a directory may have sub-directories, a project may have subprojects. The difference is that a file name in a project represents all revisions of the file—this is known as a *revision tree*—and is also a pointer to *file information* and *revision information.* File information is descriptive text that applies to all revisions, while revision information is descriptive text that relates to a single revision. Fig. 1 illustrates the project hierarchy that will be used throughout the next chapter.

The symbol "ʃ" is used as a separator in naming projects much as the symbol ":" is used in hierarchical file names. Thus, Baseʃ is a project, BaseʃSourcesʃ is a subproject of Baseʃ, and BaseʃSourcesʃCʃ is a subproject of BaseʃSourcesʃ. As is the case with directories, the terminal separator may be omitted, e.g. BaseʃSources. Although there is a parallel concept to that of *current directory*, namely *current project*, there is no provision for a partial project name relative to the current project. That is, if the current project is Baseʃ, BaseʃSources cannot be denoted by ʃSourcesʃ.

When a project is created, what one actually creates is a directory whose name is the project name. This directory always contains two files, one called *__CurUserName* that is invisible to the finder but shows up in some dialogue windows, and one called *ProjectorDB* that contains all of the project data. If the project has subprojects, then this directory will contain subdirectories (folders) that similarly house the subprojects. That is, a subproject folder will be named after the subproject, and will contain its own *ProjectorDB* file for the subproject data.

The act of *checking out* a file to a given directory is merely that of "copying" the file from the Projector Database to the directory in question. "Copying" is in quotes because in actuality the projector software may be synthesizing the file dynamically from a set of differential revision data. A checked out file contains a resource named *ckid* which identifies it as a file produced by Projector. This resource contains, amount other things, the file's revision number, whether or not the file is write-protected, and the text of the revision information.

Files may be checked out as *read-only* or as *modifiable*. If the Projector Database is accessible to multiple users, e.g. on a server, then many users may simultaneously check a file out, but only one user at a time may check it out as modifiable.

A file which has been checked out as modifiable may be *checked in* after modification. This enters the modified text as a new revision of that file in the Projector Database.

A *checkout directory* is a directory that has been associated with a project (or subproject) by execution of the command CheckOutDir. This association is private to the user and vanishes when the current MPW session ends. It may also be modified during an MPW session. The association defines the directory into which the files of the project will be checked out by default. There are no restrictions on this association. The seven projects shown in Fig. 1 may be checked out to seven different directories that bear no relationship to each other, may all be checked out to one directory, or (most commonly) may be checked out respectively to a directory structure that matches that of the project hierarchy.

A *name* is an identifier that can be attached to a group of specific file revisions, but only to one revision for any given file. It is used in commands as an pseudonym for this group, most commonly for rapid selection of the revisions belonging to a particular release. Names may be *private* or *public*. Public names become project attributes and are automatically available to all users. Private names are available only to the user who defines them, and last only for the duration of the MPW session.

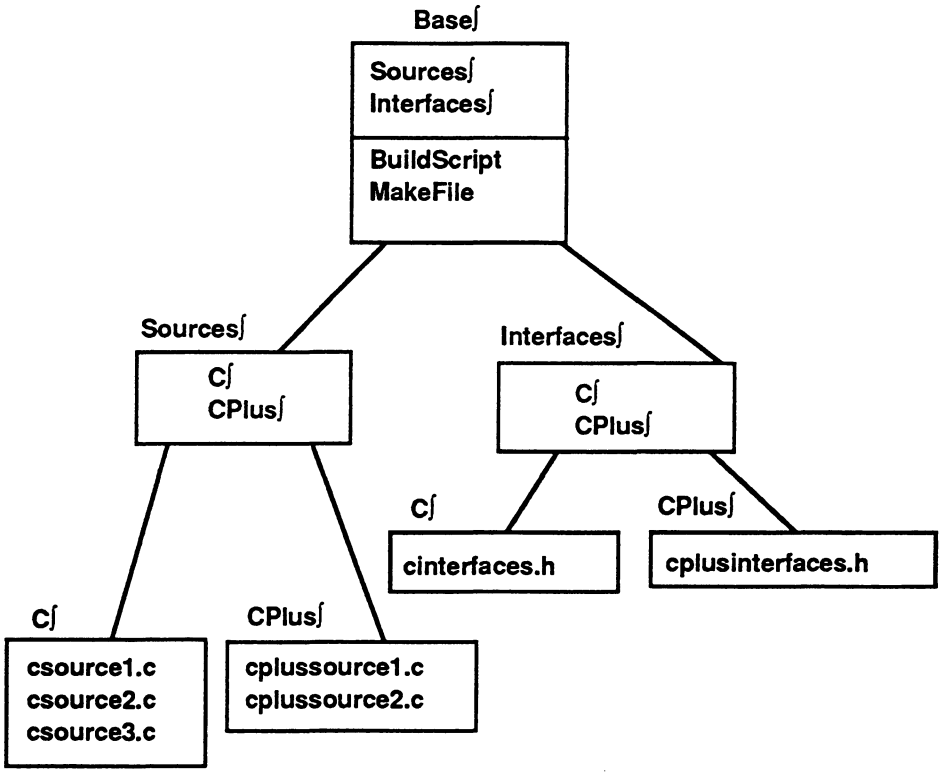


Fig. 1

Project Creation

Let us construct the project hierarchy of Fig. 1. Under the menu item "Project" in the MPW menu bar we can select the item "New Project." The "Project Menu" is illustrated in Fig. 2.

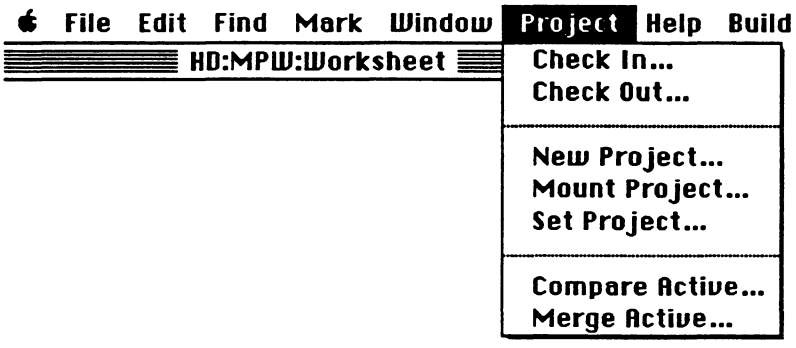


Fig. 2

We obtain a special Projector window. It is a bit like a Commando window, but it stays open until one clicks on the "close" box and it can be moved about on the screen. On the left side is a typical file

selection sub-window. We wish to establish this project inside of an already-existing directory called "ProjectDemo," and therefore navigate in the standard way until this directory is selected, giving the window as illustrated in Fig. 3. You can see it already contains two directories, *Documents* and *Examples*, which have nothing to do with the proposed project. The user enters the name "Base" and the comment respectively into the boxes labelled "Project Name" and "New Project comment." The User field is automatically set to the value of the MPW variable "user."

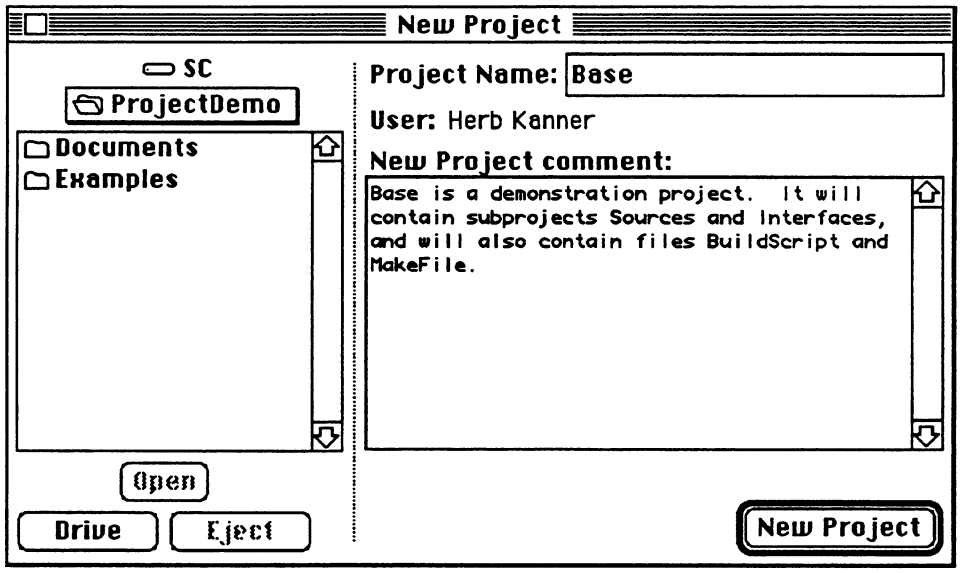


Fig. 3

After pressing the "New Project" button, the window looks like Fig. 4. The directory containing the empty database for the project Base has been created; its name can be seen in the left-hand window.

Now, clicking on the item "Base" in the left-hand sub-window will activate the "Open" button. Clicking on the latter or double-clicking on "Base" will make it the current directory, and now the new projects "Sources" and "Interfaces" can be created as subprojects of "Base" in the same way as "Base" was created. Fig. 5 shows the window after this has been done. The Projector files belonging to "Base," namely "__CurUserName" and "ProjectorDB" are visible but dimmed. Similarly, the two subprojects "C" and "CPlus"that are subprojects of both Interfaces and Sources can be created. Fig. 6 is the Finder window for Base, showing the folders (directories) for the subprojects Source and Interfaces and the actual Projector file for Base: ProjectorDB. Fig. 7 contains the Finder windows for both Sources and Interfaces, showing their ProjectorDB files and their subprojects.

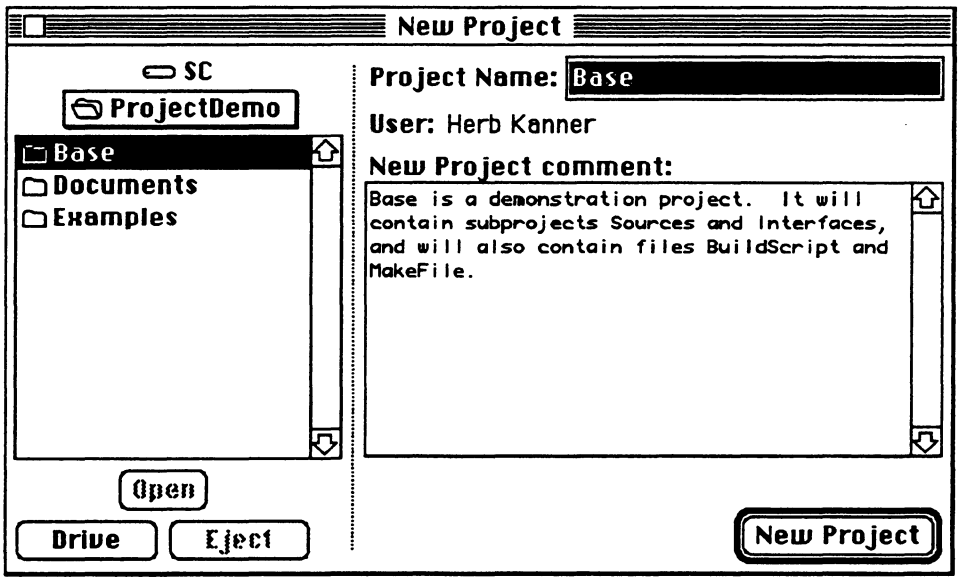


Fig. 4

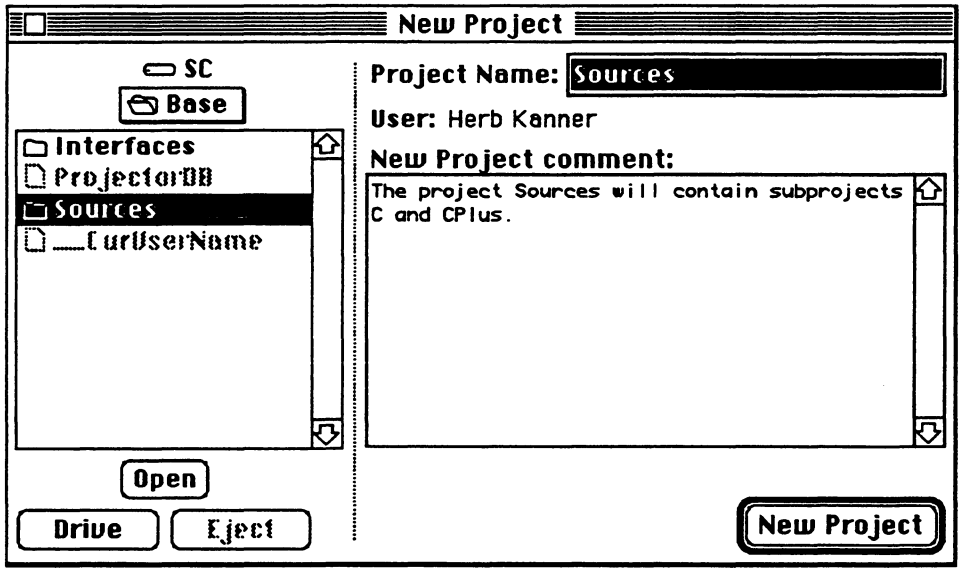


Fig. 5

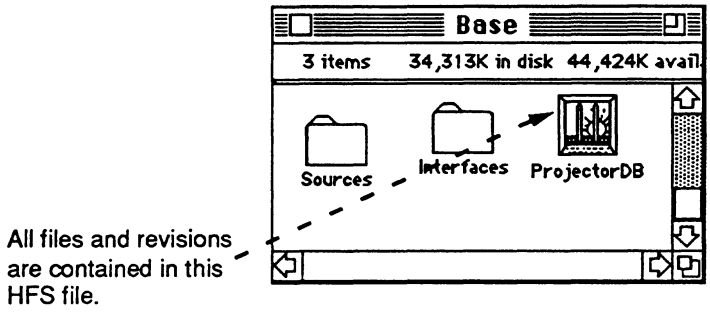


Fig. 6

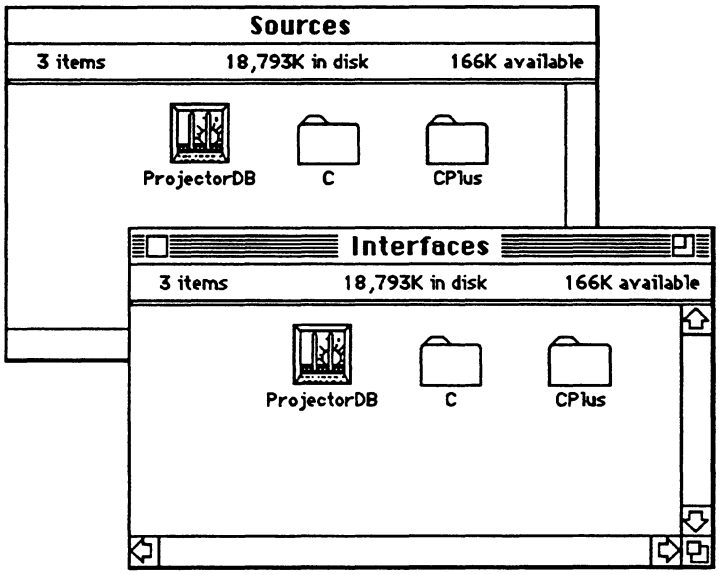


Fig. 7

Before proceeding to the use of Projector for the storage, updating, and retrieval of files, it is necessary to discuss the commands MountProject and Project. What we have done so far is to create a skeletal Projector database, that is a hierarchical project structure and implicitly to *mount* all of the projects. Mounting a project is analogous to opening a file. It makes the MPW Shell aware of the project and makes the project data available to the user. Prior to mounting, projects exist only in file storage media. The next time MPW is initiated, the projects whose creation was illustrated in Figs. 1–4 will have to be mounted again before they can be accessed. The simplest way to do this is with an unadorned MountProject command whose parameter is the project's directory:

```
MountProject SC:ProjectDemo:Base
```

Mounting a project automatically mounts all of its subprojects, so only this one command is required. One can also use Commando, and in fact the Commando window for MountProject can be activated directly from the "Project" item in the MPW menu bar. The radio button should be left at its default value "Generate MountProject Commands." When the button labelled "Project Location" is pushed, the choice "Select a project to be mounted..." will cause generation of the normal dialogue window for directory selection. The one selected will become the argument for the MountProject command.

A related command that is worth introducing at this point is Project. Analogous to the file system concept "current directory" is the Projector concept "current project." The Project command is used to set the current project, and, again in analogy to the behavior of Directory, when given with no parameters it returns the value of the current project. A special Projector window is available from the menu bar, called "Set Project... ." It displays all mounted projects in project hierarchy notation (using ʃ); the selected one will become the current project.

The MountProject command can also be given with no parameters. In that event, it returns the names of all currently mounted "root level" projects. The default behavior when doing this is to return each name in the form of a complete MountProject command, i.e., to precede the project name with the word MountProject. Figs. 8–10 illustrate some uses and variants of the Project and MountProject commands.

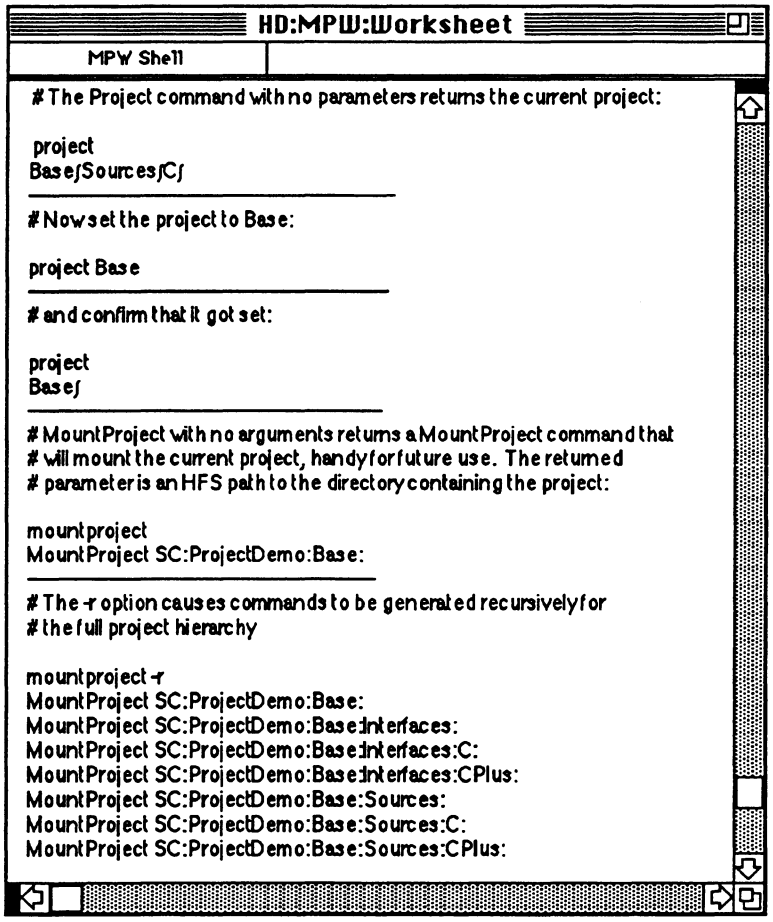


Fig. 8

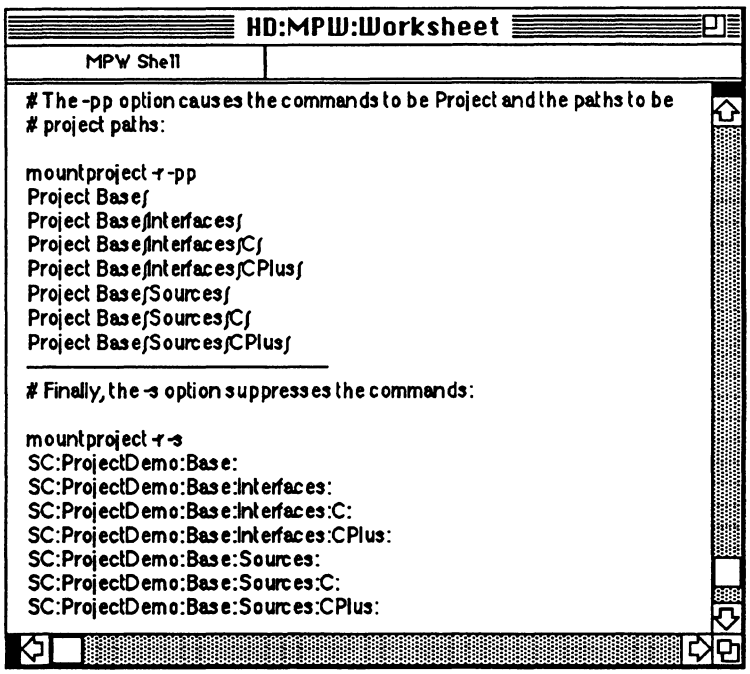


Fig. 9

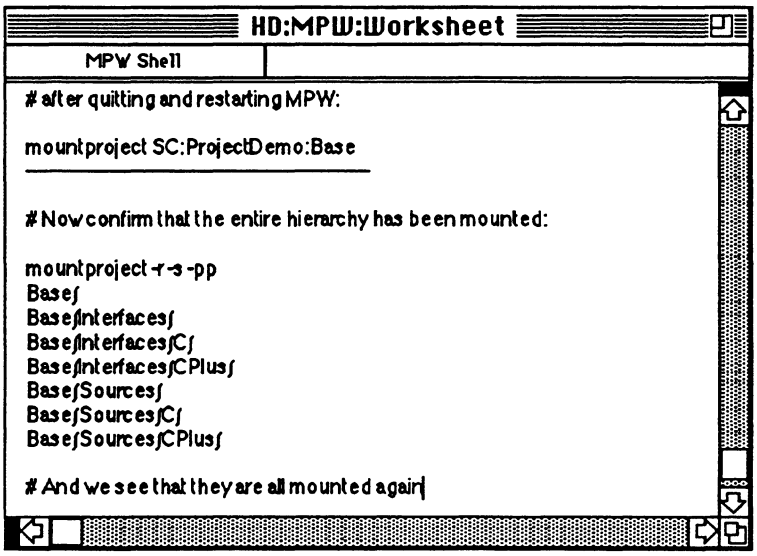


Fig. 10

This might be a good moment for the user to experiment with the "Mount Project" and "Set Project" items in the Project Menu. Mount Project is a conventional Commando window. Set Project creates a window (Fig. 11) that lists, for selection, all mounted projects. It is used in much the same way as the pop-up list of directories in the MPW Directory Menu.

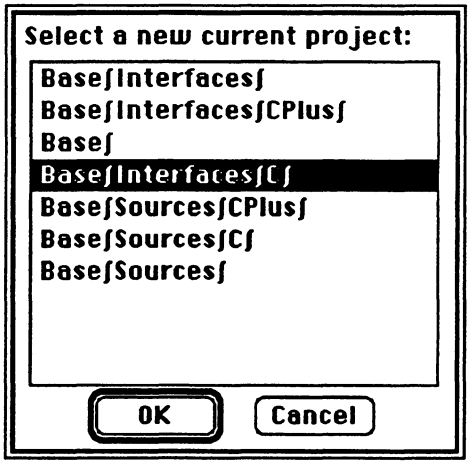


Fig. 11

Relating Directories to Projects

Now that a tree of projects has been created, we wish to put some files into them. Let us make a simplifying assumption which corresponds to the most probably desired organization: that the directory structure into which the working copies of files are to go should be an exact replica of the project structure just created. The first step is to create the directory structure that will in time house the files when they have been checked out of a project. This is done with a command called CheckOutDir. In its simplest form it takes two parameters, a project and a directory. The effect of executing this command is a bit modal: it sets a default so that subsequent CheckOut commands addressed to that project copy the files to the named directory unless another directory is explicitly named in the CheckOut command itself. A side effect of CheckOutDir is that if the directory does not exist, it creates it. A lovely option to this command is -r; with this option, sub-directories are created corresponding to all subprojects and they are given the same names as the subprojects. As is the case with MountProject, a CheckOutDir command with no arguments creates an instance of the command showing the directory that corresponds to the current project.

For the purposes of this tutorial, we want to create a set of checkout directories that parallels the project Base. We would like to put them in the same directory that contains Base, namely :ProjecDemo. Since the name "Base" has been already used, we will call the root of the checkout tree "Baseckout." Fig. 12 illustrates the use of the recursive CheckOutDir command.

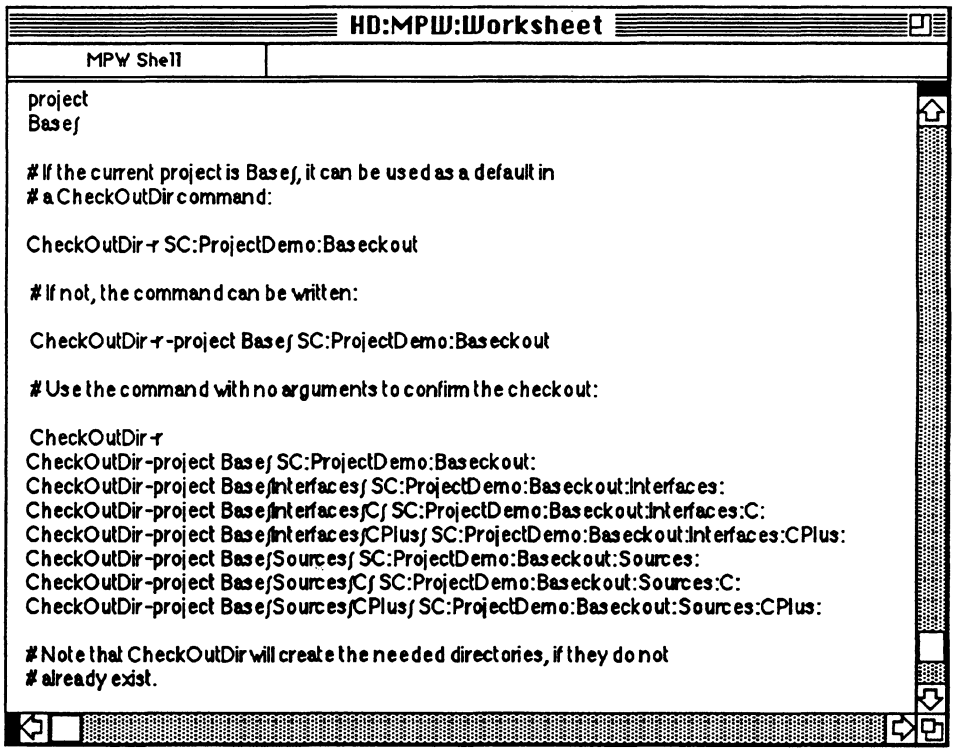


Fig. 12

Checking Files In and Out

The most used and most complicated windows specific to Projector are Check In and Check Out. Check In moves file data from normal file storage into the Projector database. Check Out copies files from the database to normal file storage. The MPW commands CheckIn and CheckOut serve the same purpose, but other than for usage inside of scripts, it is strongly recommended that the windows, selectable by choosing respectively "Check In..." and "Check Out..." in the Project item of the MPW menu bar, be used. These windows remain open until explicitly closed, and can be moved to where desired on the screen. The two windows are partially keyed to each other in that changing the current project on either one affects both windows. Most of the illustrations in this section show both windows, although in practice usually only one is opened at a time. The examples reflect the directory/project structure created in the previous chapter.

Let us now assume that the required files have been written in the appropriate directories, as shown in the following list:—

:Baseckout:
    MakeFile
    BuildScript
:Baseckout:Sources:C:
    CSource1.c
    CSource2.c
    CSource3.c
:Baseckout:Sources:CPlus:
    CPlusSource1.c
    CPlusSource2.c
:Baseckout:Interfaces:C:
    CInterfaces.h

:Baseckout:Interfaces:CPlus:
CPlusInterfaces.h

Opening the Check In and Check Out windows, we get the display shown in Fig. 13.

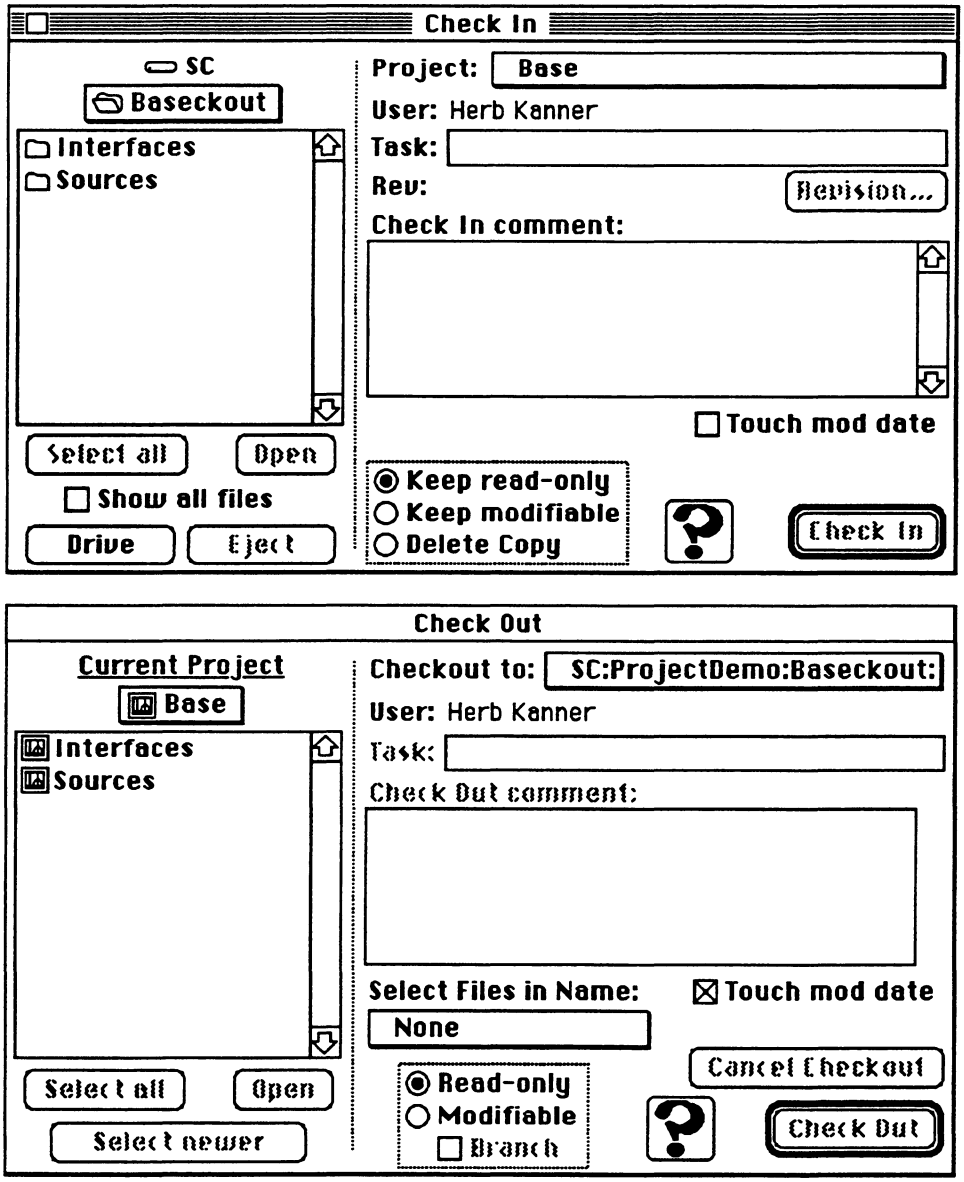


Fig. 13

At the upper right in the Check In window can be seen a button labelled "Project:". The text on the button is the name of the current project. If no project has been mounted, this text will read "Root level projects." The current project in this case is Base. If the button is pressed, the names of all projects pop up in a selectable list, with the subproject nesting indicated by levels of indentation. On the left is a display similar to the familiar dialogue used for opening files. Until a CheckOutDir command has been given, it merely displays the contents of the current directory. If the project shown in the "Project:" button has a checkout directory, then that directory will automatically be the subject of the display on the left. The

user may navigate freely within this left-hand portion of the window and select any directory and file on any mounted volume; we will return to the application of this later. On closing and re-opening the Check In window, or even by just leaving and project Base and then returning to it by use of the button at the upper right, the left-hand display will again be set to the checkout directory for Base.

Note that the Check Out and Check In windows track each other in that they both always show the same current project. The left hand display of the Check Out window permits navigation through the project hierarchy and selection of files belonging to the Projector database in exactly the same manner as the customary dialogs for Open do for directories and files. Whatever project is selected as the current project also shows up on the Project button in the Check In window and vice versa. Whatever directory appears automatically near the upper left of the Check In window (right under the name of the volume) will also appear on the button labelled "Checkout to:" at the upper right of the Check Out window. The directory can be changed independently in either of these windows. It automatically reverts to the one established by the CheckOutDir command (if such a one exists) on deselecting and reselecting the project.

Now, let us redirect our attention to the Check In window of Fig. 13. We see in the left-hand display two subdirectories of Baseckout: Interfaces and Sources. No file names are visible even though the files BuildScript and MakeFile reside in the directory Baseckout. The reason is that the Check In window by default only lists files belonging to the current project. Since the project "Base" is brand new, it does not yet contain any files. To make the file names Interfaces and Sources visible in the window, we mark the box labelled "Show all files," yielding the window shown in Fig. 14.

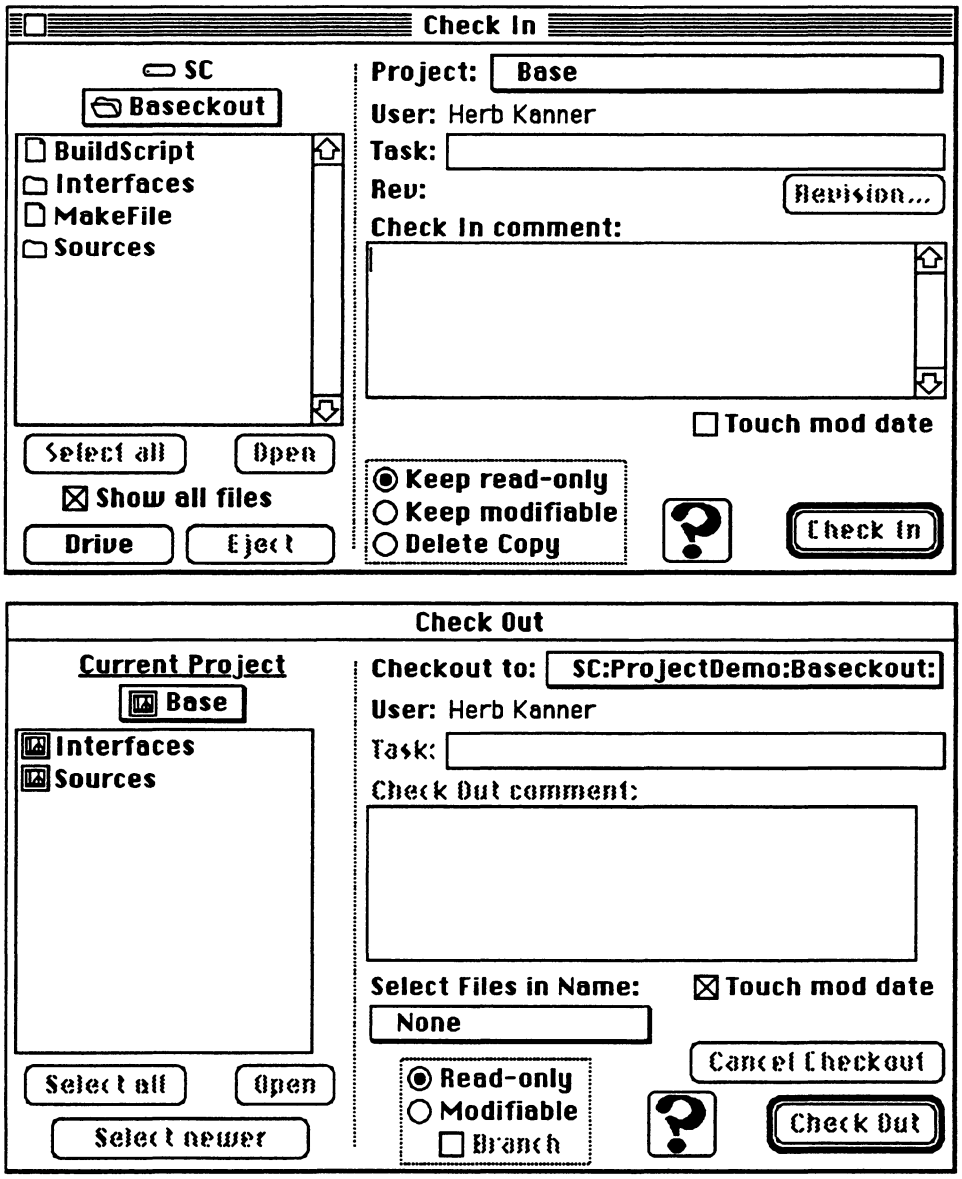


Fig. 14

The next step, of course, is to check these files in. Selecting a file and pushing the Check In button will enter the file and all ancillary information about it into the Projector database; this includes any Check In comment that has been written. After this has been done for both BuildScript and MakeFile, the windows will have the appearance of Fig. 15. This illustration also shows the Project pop-up list in the Check In window.

Observe the icons used in both of these windows for files, directories, and projects. For a detailed list of the meanings of all of the Projector icons, see the MPW Reference Manual. Because the radio buttons in the Check In window were left at their default setting (Keep read-only), the names of the two files are now gray, indicating that because they were checked in, and because the copies in the directory Baseckout are now read-only, they clearly should not be checked in again. The icon indicates that they are read-only, and if one now opens the copy of the file in the checkout directory, one will see that same icon in the upper left-hand corner of the file window, and indeed, any attempt to write to that file will meet with failure.

Notice that each window has a button labelled with a great big question mark. Pushing this button changes the right side of the window to an information window and modifies some of the other buttons. Doing this to both windows yields the pair shown in Fig. 16. Note that the window titles have been changed to remind one that they are now yielding information. The same button, now labelled "Done," when pushed again causes the window to revert to its original state.

Selecting and "opening" a file in the Check Out or Check Out/Information window demonstrates the next level of access, that of the file revision. Fig. 17 shows the appearance of the Information windows after opening MakeFile. We see an icon labelled "1," the revision number of the only existing revision. As the file is checked out for modification and checked back in, additional such icons will appear, corresponding to all existing revisions. This permits easy fetching of earlier revisions when desired. The radio buttons labelled "Latest Revision Info" and "File Info" permit respectively the choice of a comment that applies to the specific revision selected and a comment that applies in common to all revisions of the file. The term "latest" in the label is not exactly accurate. It reflects the fact that the default selection will, in fact, be the latest revision.

At the start of this section, it was assumed that the original revisions of the files to be checked in already exist in their correct checkout directories. This assumption was made for convenience. By first setting the desired project in the box at the upper right in the Check In window, checking the "Show all files" box, and then selecting desired files for that project by use of the left-hand sub-window, it is easy to check in files from anywhere.

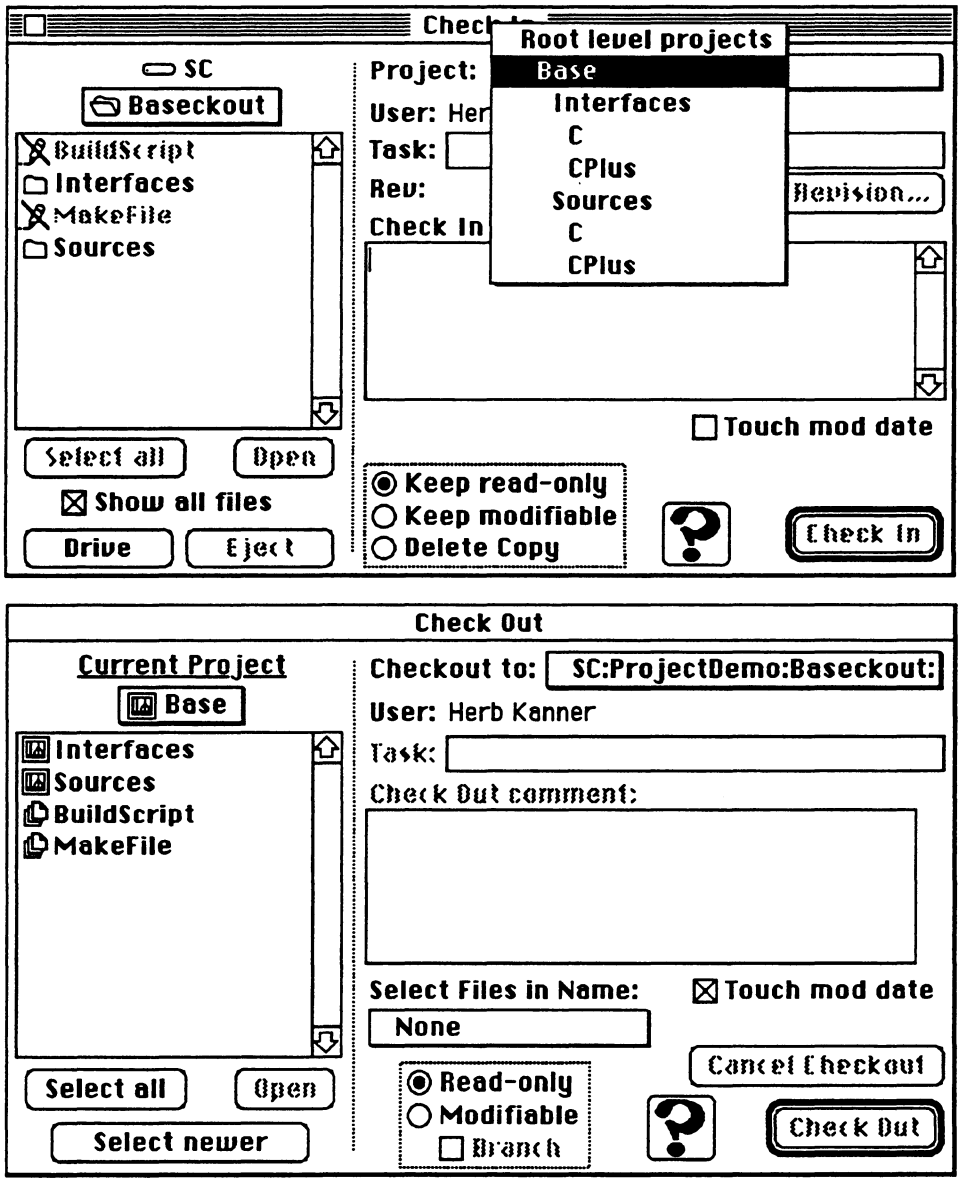


Fig. 15

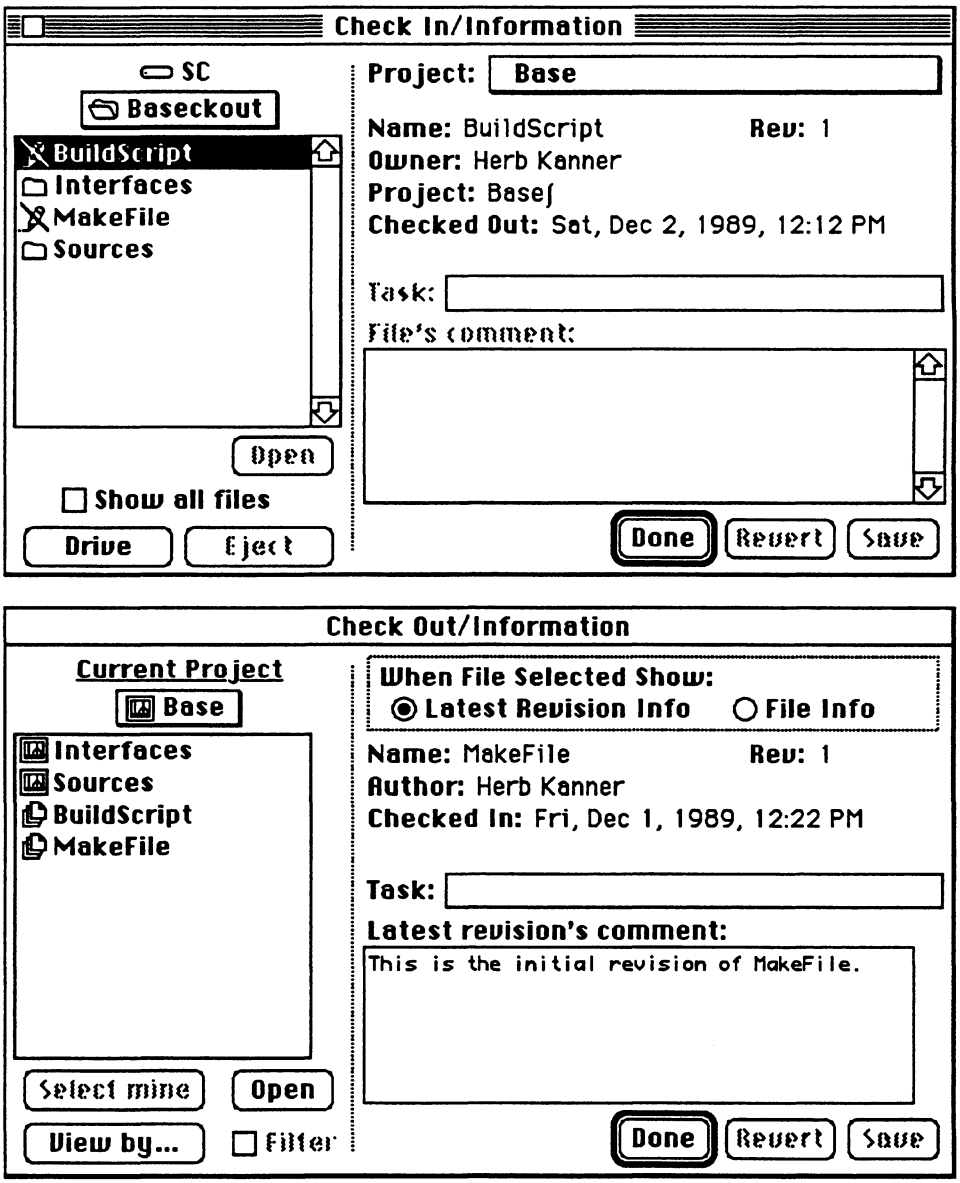


Fig. 16

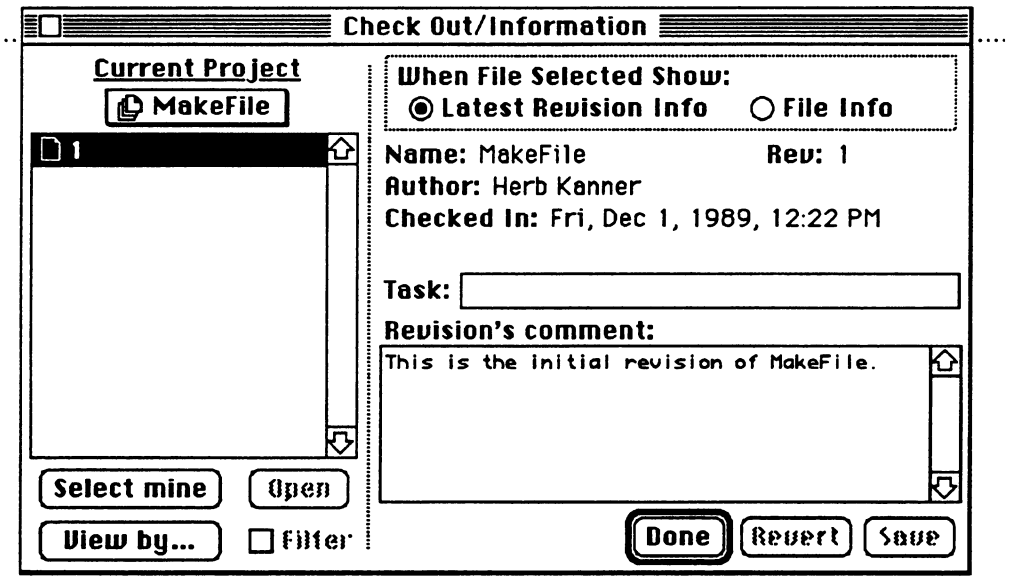


Fig. 17

Let us next assume that original revisions of all the files shown in Fig. 1 have been checked into their proper projects. It is now our intent to make some constructive changes to the file CSource1.c. We go to the Check Out window, navigate in the left-hand area until the current project is Base∫Sources∫C∫, and select the file CSource1.c. Before pushing the Check Out button, we make sure that the radio button "Modifiable" has been pushed. After writing a comment and pushing the Check Out button, we find that the two windows have the appearance of Fig. 18. Note the icon next to CSource1.c in both windows; it indicates that the checked out copy is modifiable. If we now go to the Check Out/information window by pushing the big question mark and open CSource1.c, we get the display of Fig. 19. We see that the revision being modified is called "1+." After it has been checked back in, this number will change to 2.

This last demonstration could also have been accomplished using the Check Out window, instead of using the Check Out/Information window. However, the file CSource1.c appears dimmed in that window. This is correct. Because it has been checked out for modification, any other attempt at checkout must be only as a branch. Making the file unselectable cautions the user. This precaution can be overridden by holding down the *option* key while clicking on the file name. The file can then be opened to display the revisions as before. Notice that when an "open" is forced on a file that has already been checked out for modification, the box labelled "Branch" in the Check Out window is automatically marked.

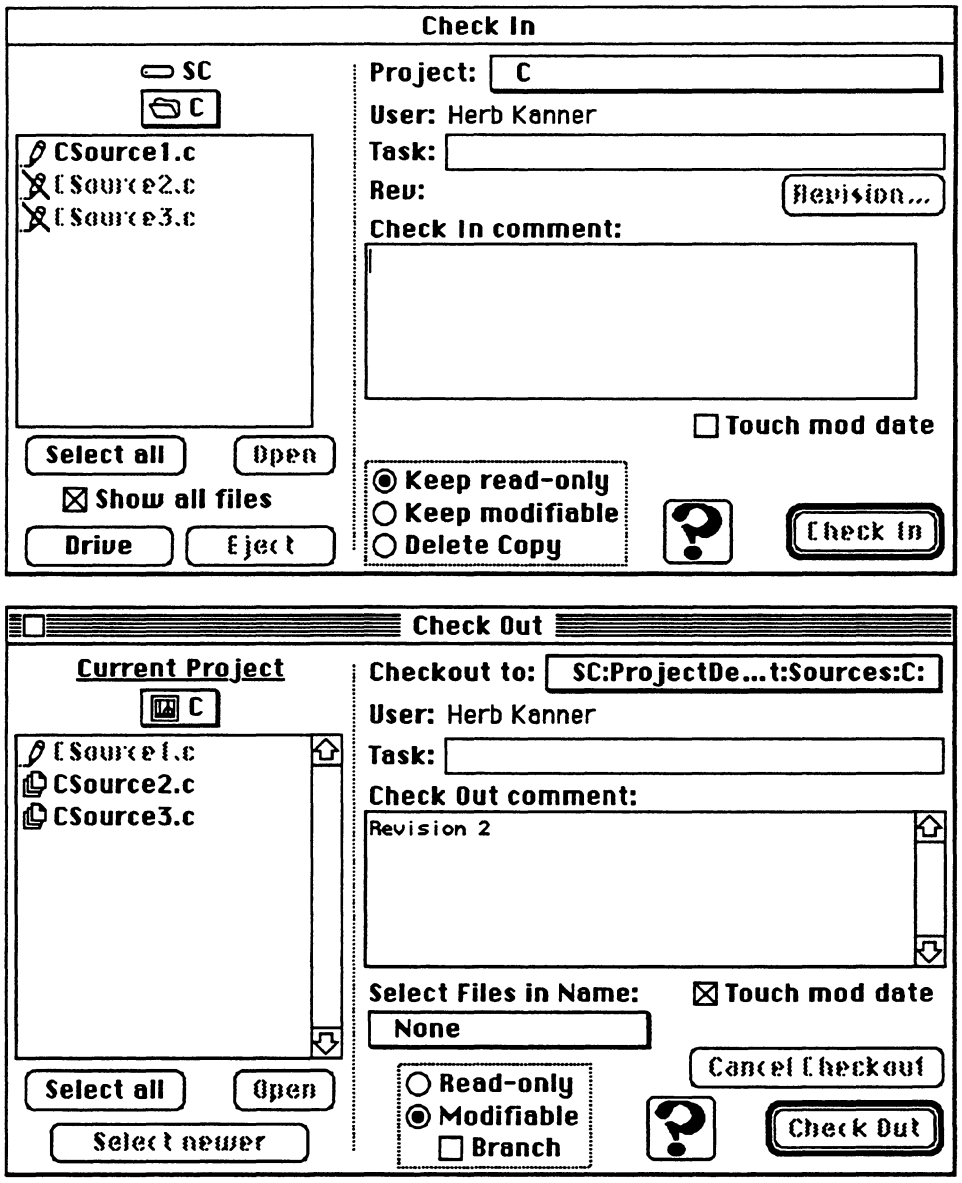


Fig. 18

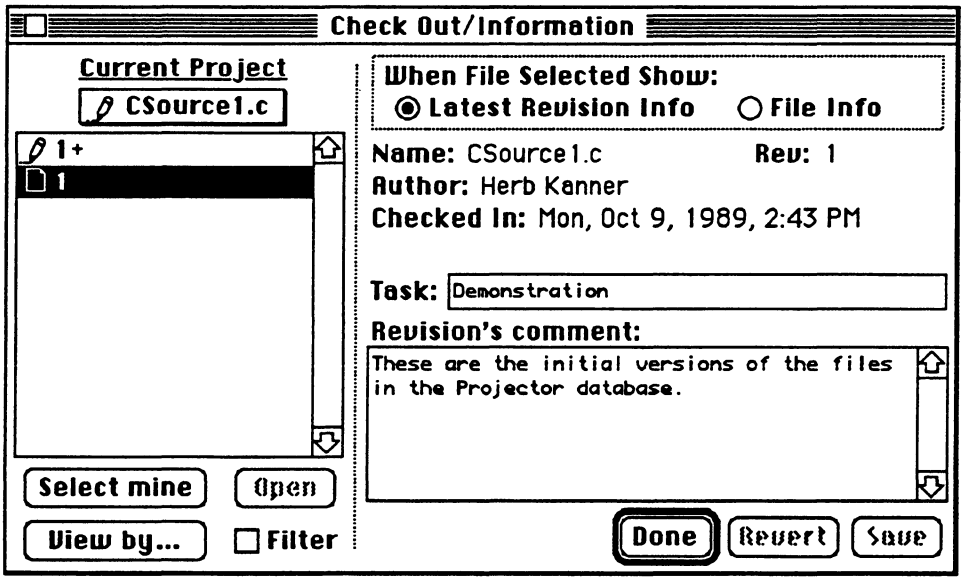


Fig. 19

Branching

The normal sequence of checking out a file as modifiable, editing it, and checking it back in produces what is called "the main trunk," a series of revisions that are numbered in sequence: 1, 2, 3, ... The button labelled "Revision..." in the Check In window may be used to create gaps in this sequence. That is, if revisions 1 through 4 exist, so that revision 5 would be created next, the use of this button makes it possible to name the next revision with an integer greater than 5. Often, it is desirable to pursue a parallel development while work on the main trunk proceeds. The revisions belonging to the parallel development are said to be on a *branch*. Methods will be shown later for merging files developed along a branch back into the main trunk. The notion of branching is recursive; a branch may be created that diverges from an already existing branch and this may be done to any desired depth. Multiple branches may be taken from the same revision. There is a numbering scheme for branch revisions which enables the user to visualize the tree, knowing only the revision numbers. This branching capability accounts for the term *revision tree* to describe the set of revisions of a file.

Branching from the latest revision is simple. If, for example, the current revision of CSource1.c is Revision 3, then all that is needed is to click on the Branch box before checking out the file as modifiable. The file while it is checked out will be labelled Revision 3a+, and on being checked back in will become revision 3a1. A second parallel branch from the main trunk would be labelled 3b1 after check in. If 3a1 is checked out modifiable, revised, and checked back in, it becomes 3a2. A branch from 3a2 would become, after checkin, 3a2a1, and so on.

Branching from earlier revisions is slightly more complicated. Let us assume again that CSource1.c is up to Revision 3, and that it is desired to revert to Revision 1 and branch from there. Go to the Check Out window, press the "Modifiable" radio button, then select and open CSource1.c. The three revisions will now show, but all but the last will be dimmed. Select Revision 1 by holding down the *option* key while clicking on it. Notice that the Branch box will be checked automatically. An alternative procedure is to select Revision 1 after pushing the "Read-only" radio button, and then pushing the "Modifiable" button. After checkout, the window will taken on the appearance of Fig. 20. After the branch is checked back in, the window will be as shown in Fig. 21.

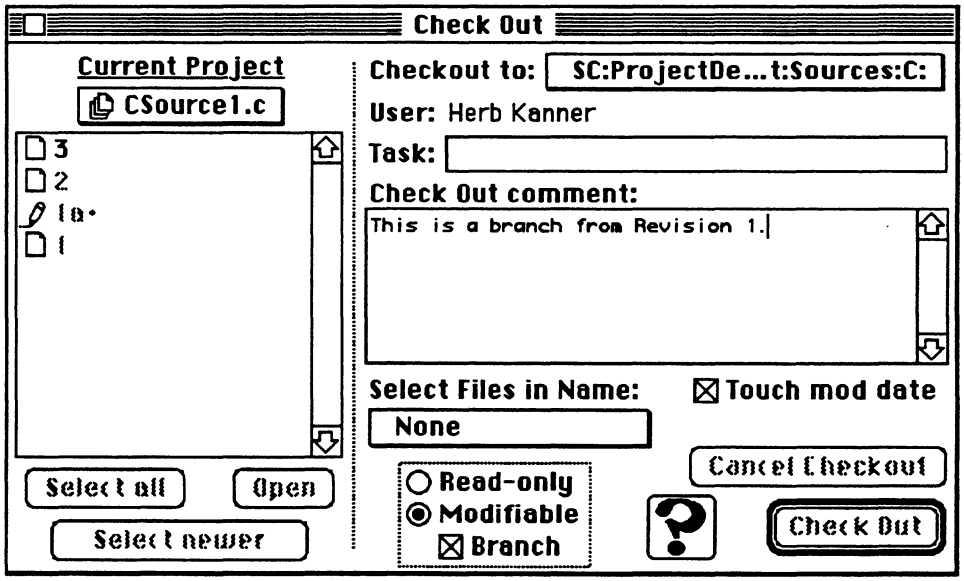


Fig. 20

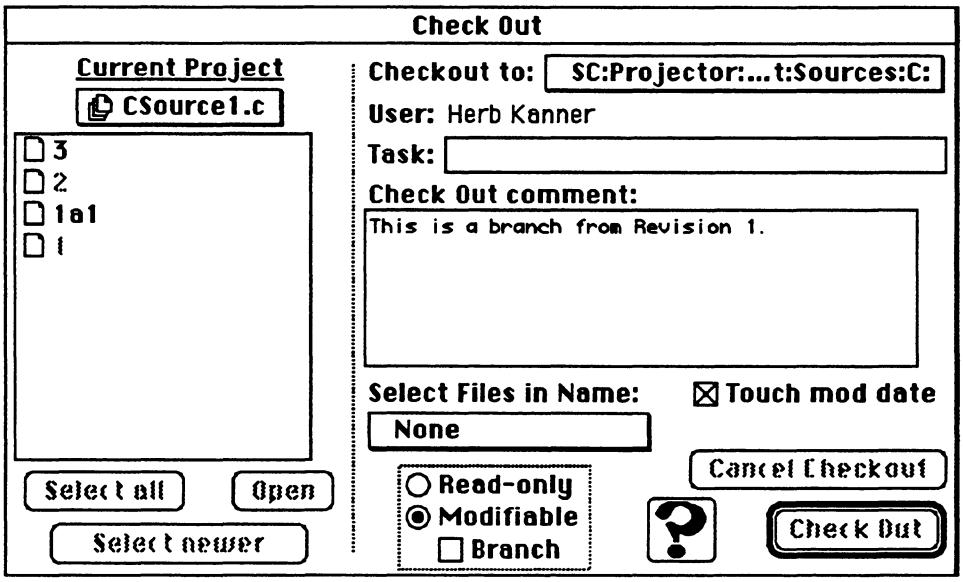


Fig. 21

Fig. 22 shows the members of a moderately bushy revision tree (the one hidden item is Revision 1), and Fig. 23 shows the same tree graphically.

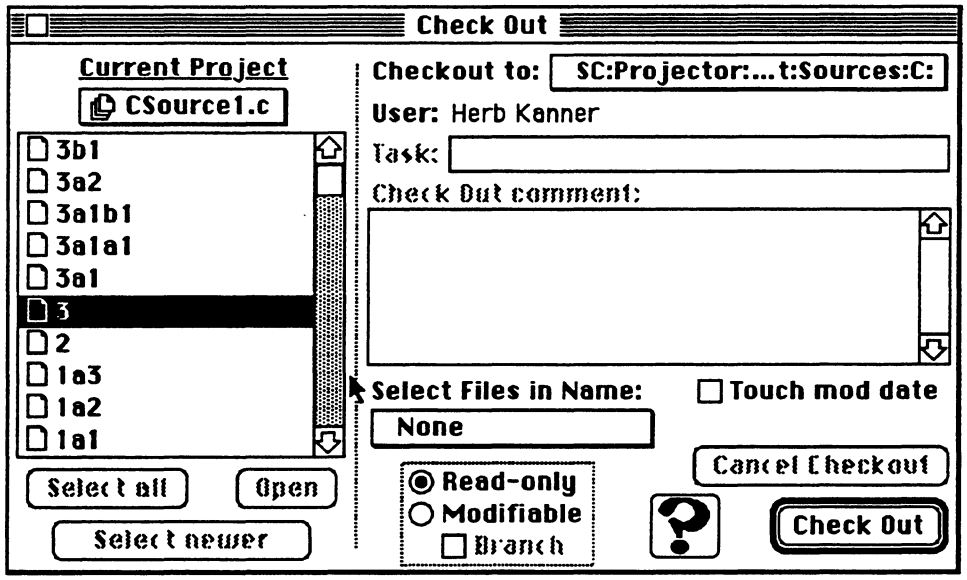


Fig. 22

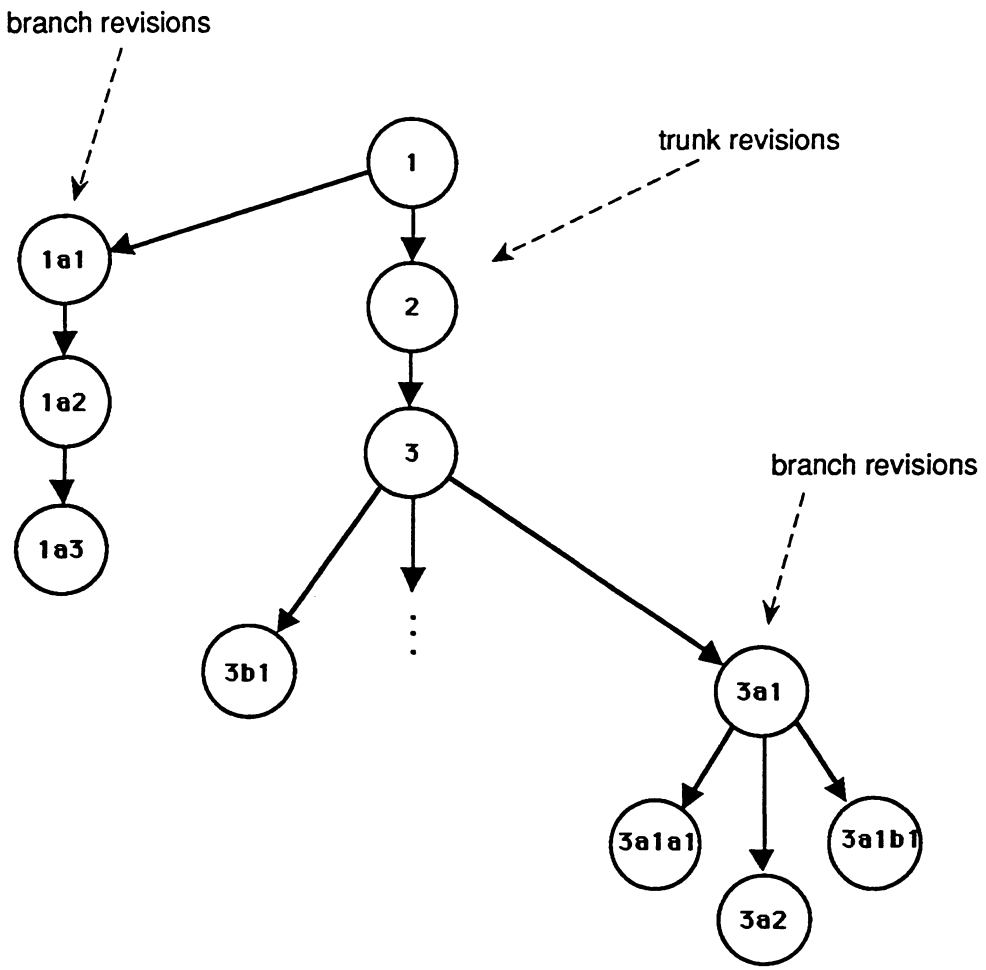


Fig. 23

Miscellaneous Buttons, Icons, and Special Keys

We are now in a position to discuss most of the remaining features of the Check In, Check Out, and New Project windows.

The boxes labelled "Touch mod date" to be found in both the Check In and Check Out windows cause the date of latest modification in the file system directory to be set, respectively, to the time of the checkin or checkout. By default, this is marked in the Check Out window and not in the Check In window. Although this can cause unnecessary revisions of this date, it guarantees an update on every checkout, meaning that tools like Make will always assume that they are being presented with a new version. If this default is not used, and more than one person is working on a file, then there is a danger that a user may check out a revised file and send it to Make without the latter program realizing that the file has been updated. If there is only one user working with a set of projects, reversing this convention and touching the mod date on checkin may be more convenient.

The button labelled "Cancel checkout" is active when a file that has been checked out for modification is selected. Pushing this button changes the status of the file to read-only and discards any changes that had been made to the file while it was modifiable.

The button labelled "Select all" causes selection of the latest revision on the main trunk of all files in the current project. It does not perform a checkout, just a selection. The button labelled "Select newer" selects, with one exception, those files for which the newest main trunk revision is not already to be found

in the user's checkout directory. The exception is any revision which is on a branch. The assumption is that if a branch has been checked out, the user intends to keep it. This button does not distinguish between the main portion of a branch and sub-branches. A revision is either on the main trunk, i.e. its revision number contains no alphabetic characters, or it is on a branch, i.e. its revision number contains one or more alphabetics. If the *option* key is depressed while the "Select Newer" button is pressed, the selection action is modified so as not to select any revision whatsoever of a file unless a copy of the file already exists in the user's checkout directory. This is equivalent to using a written CheckOut command with the **-update** option. The idea is: select file revisions for checkout by the same criterion as "Select newer," but do not check out revisions of any files that have not already been checked out. Just update the files that the user has checked out.

Multiple selection of a subset of the files shown in a Check In or Check Out window can be done in two ways. If the *shift* key is depressed while a second selection is made, then the previous selection is retained and all intervening names are selected, i.e., a contiguous set of names is selected. If disjoint set of names names are desired, then the *command* key should be depressed while making the selections after the first one. The "action" button of the Check In, Check Out, and New Project windows is keyboard activated by pressing the *enter* key. This is required because keystrokes, including that from the *return* key, are sent to the comment field in all three of these windows.

Depressing the *option* key while pushing the Check Out button will cause automatic opening of the file being checked out if it is a text file.

Some less frequently encountered icons are illustrated in the next two figures. Suppose that the user manually selects a directory in a Check In window that is not the "checkout directory" for the current project. For example, this might be done if the directory contains a file which is not yet a Projector file, and the user wants to check this file in to the current project. On marking the "Show all files" box, any Projector files in the directory which do not belong to the current project will be designated by an icon bearing a question mark and will have their names dimmed. This is illustrated in Fig. 24, where the current project is shown as C (actually Base∫Sources∫C) but the user has selected the directory Baseckout. Fig. 25 illustrates one other icon that may be seen when multiple users have access to a Projector database. The user "joe" wishes to check out the file CInterfaces.h. The padlock icon indicates that this file has been checked out for modification by another user. Projector will only allow "joe" to check out the latest revision as a read-only file. If "joe" wants to do any modification, he will have to create a branch.

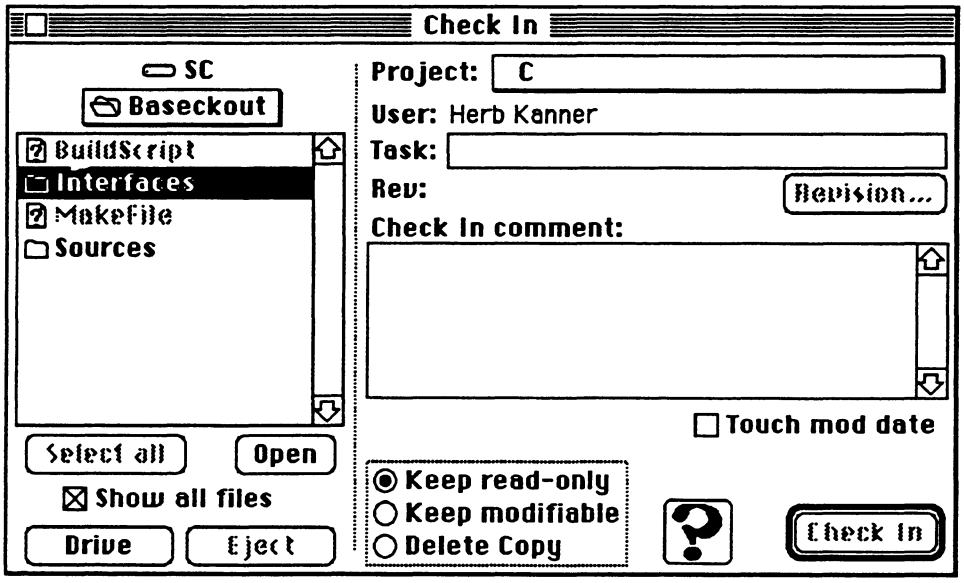


Fig. 24

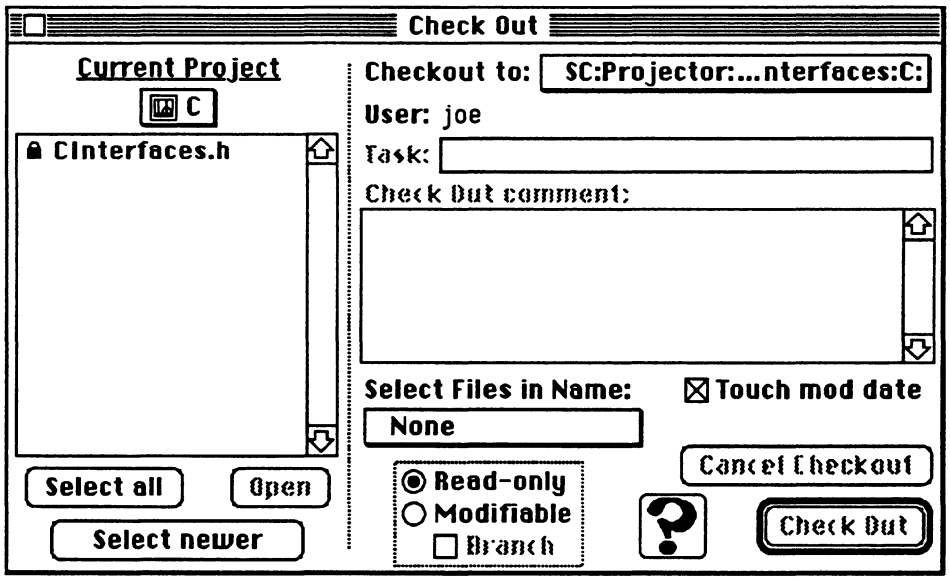


Fig. 25

Finally, the Check Out/Information window has a facility for identifying the revision of a file that is currently checked out. The procedure is as follows: Obtain the Information window by pushing the button with the big question mark. In the left-hand area of the window, select and open the file in question thus showing its revision tree. Push the "Select mine" button. If any revision of that file exists in its checkout directory, that revision will be selected in the window, thus giving the desired information.

Naming a Set of Revisions

Facilities exist in Projector for associating a name with a chosen set of file revisions. Thus, for example, the revisions corresponding to a given release, say **alpha1**, of a product can be given that name. As is illustrated in Fig. 26, the button in the Check Out window labelled "Select Files in Name:" will, when pushed, display a pop-up list of all known names. Dragging to the name "alpha1" will then cause selection of that set of revisions, enabling easy checkout of the source files for that release.

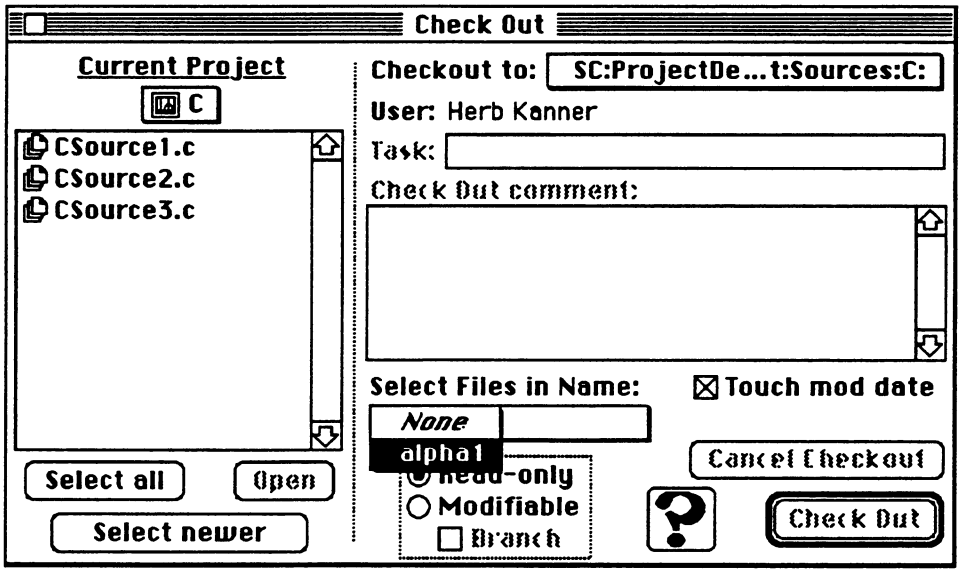


Fig. 26

The assigning of names is done with the command NameRevisions. Like several other commands already described, this command is used with parameters to associate a name with a set of revisions, and without any parameters to elicit information about existing name assignments. It is one of the more complicated commands in the lexicon. Fortunately, anything done with this command can be easily reversed with the command DeleteNames.

The three most important options of the NameRevisions command are **-public**, **-private**, and **-dynamic**. The **-public** option, which is the default, establishes a name as public and relatively permanent. It is recorded in the Projector database and will appear in the Check Out windows of all users. A private name exists only for the convenience of the user who defined it, and lasts only for the duration of the current MPW session. The only way to give it any longevity is to have the command that created it saved in a script file. Private names appear first in the pop-up list and are separated from public names by a dotted line.

The option **-dynamic** is a little harder to explain. Let us consider a couple of scenarios that illustrate when this option is and is not wanted. First we will describe the simpler case, the one where the option is not used. Suppose that the latest revision on the main trunk of every file in a project is to be used for a release. The NameRevisions command written

```
NameRevisions -project myproject -a thisrelease
```

will freeze the name "thisrelease" to the latest main trunk revisions of files in the project "myproject." The **-a** option indicates that we want all of the files in the project. The selection is static. That is, at some future time, by which many of the files may have been revised several times, the use of the name "thisrelease" will select that frozen set of revisions. The second scenario might be that one or more files in a project become obsolete. Suppose that a project has files valid1, valid2, valid3, obsolete1, and obsolete2. It has been decided that obsolete1 and obsolete2 will no longer be used. The command

```
NameRevisions -project myproject -dynamic ∂
      active valid1 valid2 valid3
```

will cause the name "active" to be a selector for the latest main trunk revisions of the three named files—that is, the latest revisions at the time the selection is made, not at the time the NameRevisions command was executed. Incidentally, if what was desired was the latest revision on branch "a" of, for example, valid2, that file name would be written in the NameRevisions command as "valid2,1a". If a revision is fully specified in a file name, e.g. "valid2,1a1", then that will be the selected revision, regardless of whether or not the dynamic option is used.

If a NameRevisions command reuses a name, the file revisions named in that command will be appended to the list of those previous associated with the name. If the intent is to replace the old set of revisions by a new set, then the option **-replace** must be used. A name can be expunged by using it as an argument of the DeleteNames command.

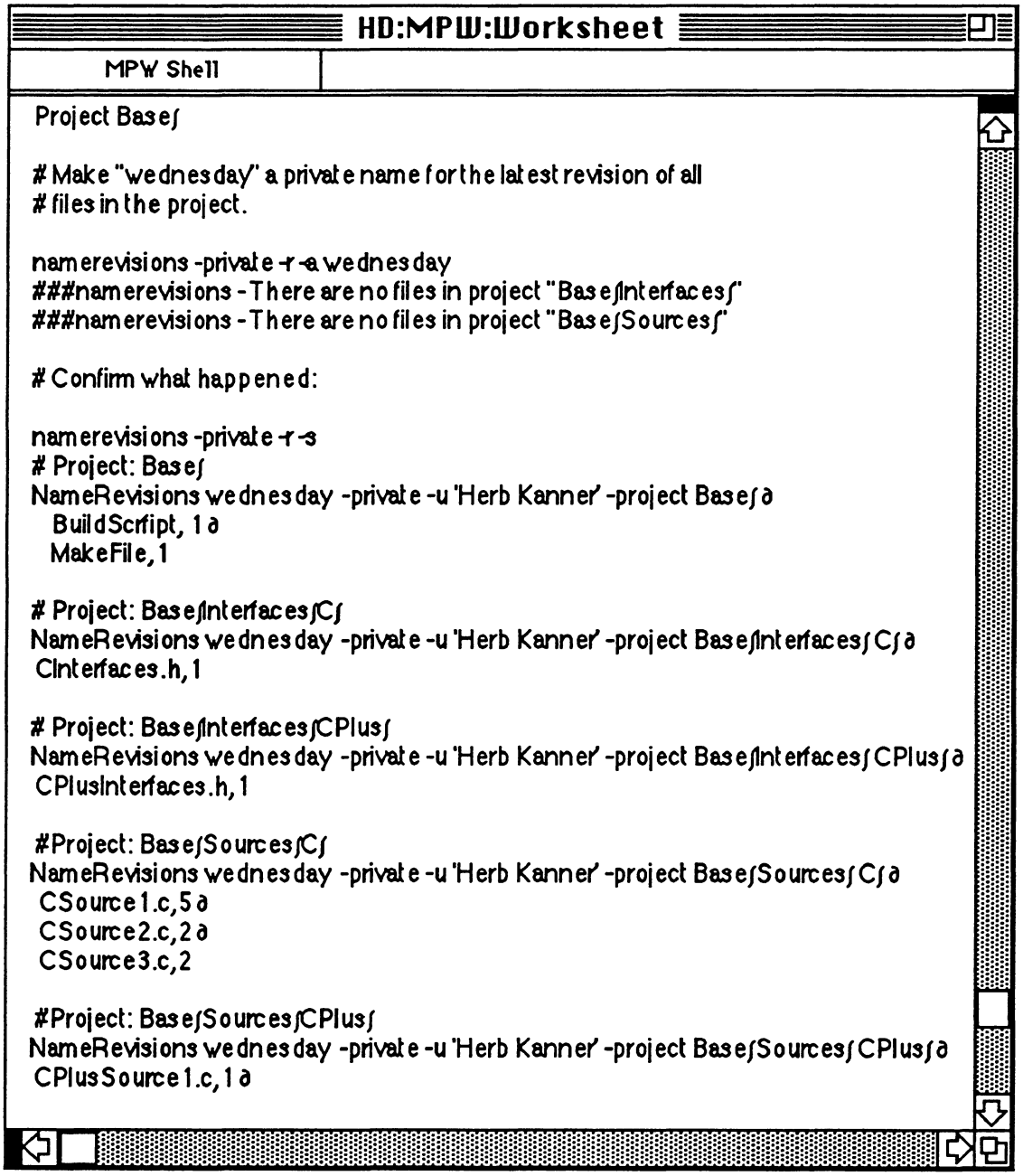


Fig. 27

The application of NameRevisions is demonstrated in the next few illustrations. In Fig. 27, using the **-r** (recursive) option, the name "wednesday" is associated with the all files in base∫ and all of its subprojects. Note that two diagnostic messages are emitted about the subprojects that do not contain files. As in MountProject, omission of necessary parameters causes that command to emit information, but includes the command name itself for possible future use. So, using no parameters, the execution of the command with the **-r** option causes printouts of the NameRevisions commands for each level of the project hierarchy that would associate the name with the applicable revisions. The effect is to get a nice list of the exact revisions that would be selected by use of the name.

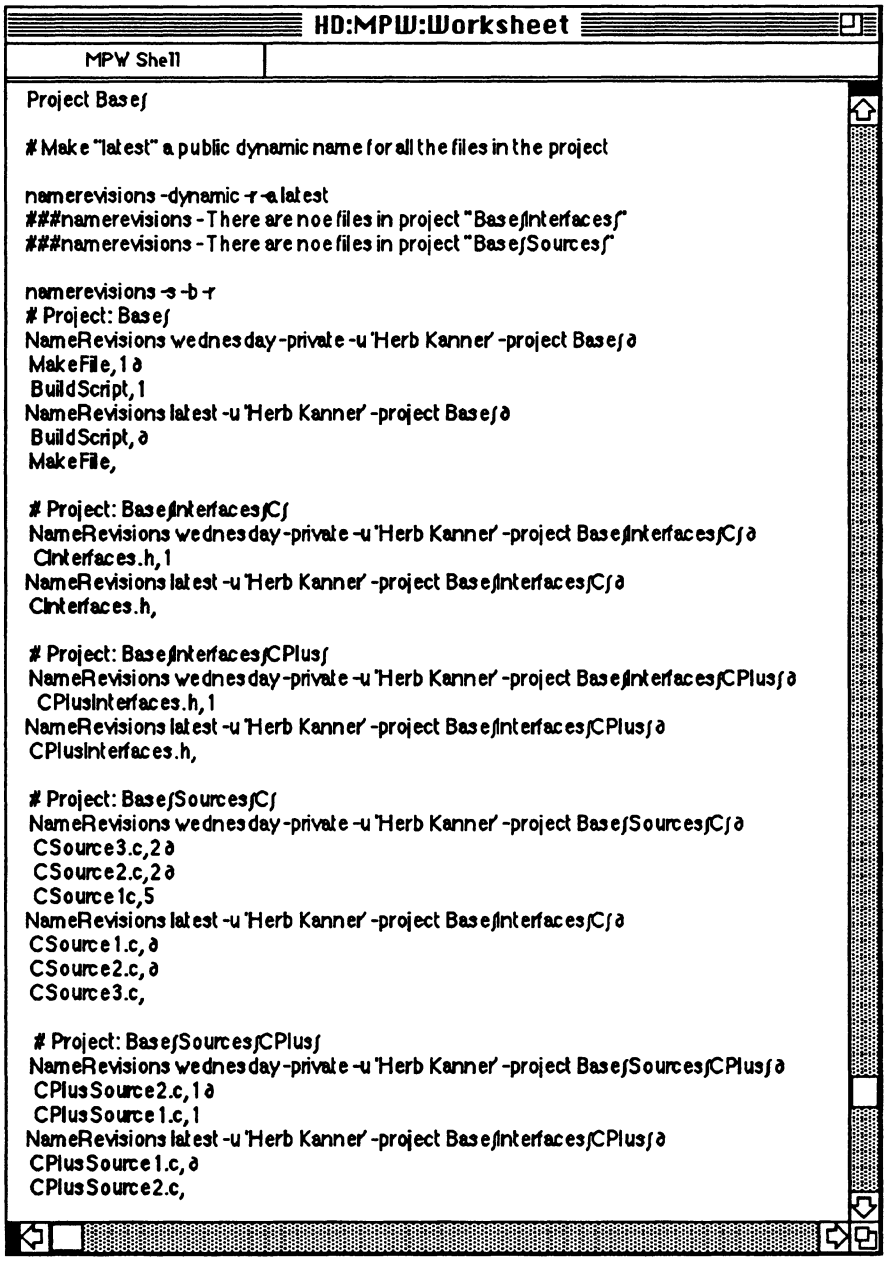


Fig. 28

In Fig. 28, the exercise of Fig. 27 is repeated, but this time the name "latest" is defined to be a dynamic name. The parameterless NameRevisions command is now given an additional option, **-b**, which forces it to list both public and private name assignments. We get to see now the files associated with the public name "latest" and with the private name "wednesday." Note that the files associated with the dynamic name are indicated by the file name followed by a comma, but with no revision number. This is because a file selected by a dynamic name is automatically the latest main trunk revision, so it is not

usually desired to see the revision number. If the revision number is wanted, the option **-expand** will force it to appear as is demonstrated in Fig. 29.

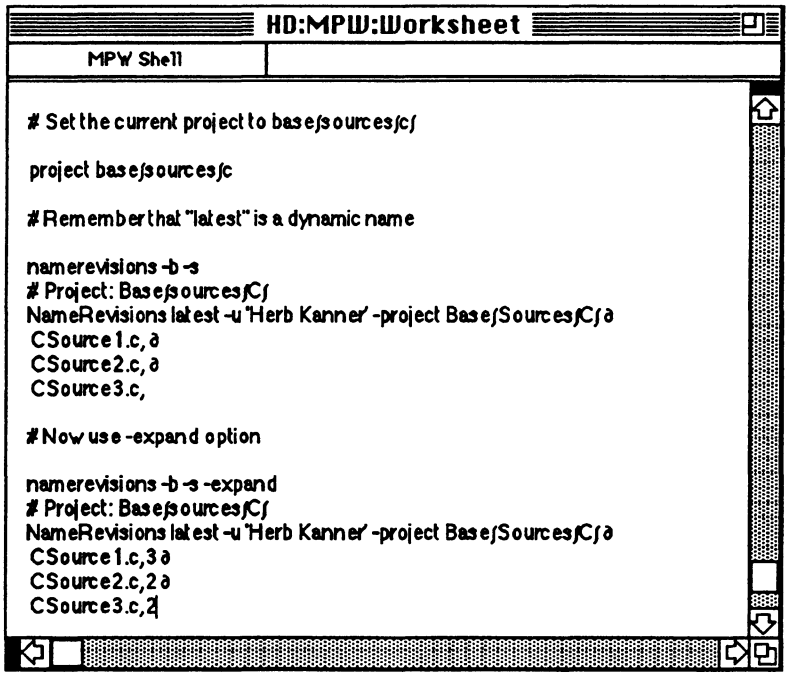


Fig. 29

If You Must Be Different

The demonstrations given so far make the checkout directory mimic the structure of the project itself. Each individual user of the project is free, however, to structure the checked out files in any desired way. For example, the commands:

```
CheckOutDir -project Base∫Sources∫C sc:projector:CFiles
CheckOutDir -project Base∫Interfaces∫C sc:projector:CFiles
CheckOutDir -project Base∫Sources∫CPlus sc:projector:CPlusFiles
CheckOutDir -project Base∫Interfaces∫CPlus sc:projector:CPlusFiles
```

will create directories ":CFiles:" and ":CPlusFiles:", will cause all files in Base∫Sources∫C∫ to be checked out in the directory ":CFiles:", and so on.

Comparing Revisions and Merging Branches

We discuss here the last two items on the MPW menu bar under "Project." The first, named "Compare Active...", calls the script CompareRevisions. The second, named "Merge Active...", calls the script MergeBranch. Both of these scripts do a small amount of housekeeping and call on the MPW script CompareFiles for the main body of the task. When "Compare Active" is invoked—the active window must

be a checked out projector file—a selector window will appear naming the revision number of the active window and listing all other revisions so that one can be chosen. When the desired revision has been chosen, CompareFiles goes into action, and its windows display all the differences between the two revisions. "Merge Active..." has a similar set of mechanisms and some constraints. The active window must be a branch revision. The other window, implicitly chosen, is the latest main trunk revision, checked out for modification. The mechanisms of CompareFiles permit the user to find the differences and selectively to copy and paste material from the branch to the main trunk. This is the method to be used when work on a branch proves to be fruitful and it is desired to incorporate that work into the main line of the project. For details on the behavior of CompareFiles, see Volume 2 of the MPW Reference Manual.

A particularly simple case of the application of Merge Active is of frequent occurrence, and is worth examining in detail. Suppose we check out Revision 10 of a file for modification, apply a set of changes applied, and then check it back in. After some thought, we decide that the current revision, namely Revision 11, is absolutely worthless and that we would like to revert to the previous revision. Unfortunately, there is no way to delete a single revision from the top. The best that we can do is to create a Revision 12 that is a duplicate of Revision 10. Here is the quickest way to do this. In the Check Out window, "open" the file to show the revision list, chose Modifiable and Branch and then select Revision 10 while holding down the *option* key. Press the Check Out button. A branch revision from 10 will be checked out. Now open the file, so that the active window contains this revision. Select "Merge Active..." from the Project menu. When the machine has settled down, the following will be seen: At the top of the screen will be two windows, side by side. One of them will be read-only, and will be the branch 10a1. The other will be read-write, and will be the modifiable Revision 11+. (Revision 11+ becomes Revision 12 when checked in.) Inspection of the Check Out window will confirm the status of these windows. Now, do a "select all" operation, either via the Edit menu or from the keyboard, on both of these windows, and do a copy and paste of the entire branch window into the modifiable window. Next, select the pop-up item "Done" from the "Compare" menu item. This will dispose of the two windows. Finally, do a check in of the file, creating Revision 12.

Miscellaneous Goodies

Projector commands not yet considered are:

DeleteRevisions
ModifyReadOnly
OrphanFiles
ProjectInfo
TransferCkid
UnmountProject

These are discussed briefly where the writer feels that some explanation is useful. Some of them are not mentioned at all on the grounds that the material in the MPW manual is adequate.

DeleteRevisions does not do what one might expect. It cannot usually be used to expunge a mistake. Its purpose is to delete large revision sets that are no longer wanted because of obsolescence. So, the alternatives are: with the **-file** option, it will delete all revisions of the named file from a project. It will be as if the file had never been there. Without the file option, it deletes all revisions that are older than the named one on its branch, or an entire branch (see below). The obvious purpose is to get rid of stuff that is so old that no one will ever again want to see it. So, for example,

```
DeleteRevisions -project projʃ file.c
```

will delete all revisions of file.c prior to the latest one on the main trunk. If the parameter is `file.c,2a3`, then all revisions on branch 2a prior to 2a3 will be deleted. If the parameter is `file.c,2a`, then all of branch 2a will vanish.

ModifyReadOnly is an emergency kind of command. Suppose a file has been checked out read only, and then the Projector database becomes temporarily unavailable. A typical situation causing this would be that the database is on a server, and the checkout is to a portable medium which the owner takes home. At

home, a decision is made to edit this file. ModifyReadonly will remove the read only restriction from the file. This change can be confirmed by that fact that the icon in the lower left corner of the file window changes. The solid line that crosses the pencil becomes a dotted line. When this file next confronts the Projector Check In window, this same modified icon will appear next to the file name, and it will be possible to check the file in as a new revision.

OrphanFiles is used to remove completely the Ckid resource from a file, so that the file is no longer recognized by Projector.

Command Syntax

```
CheckIn -w | -close | ([-u user] [-project project] [-t task]
      [-p] [-cs comment | -cf file] [-new | -b] [-m | -delete]
      [-touch] [-y | -n | -c] (-a | file…))

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

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

CompareRevisions file

DeleteNames [-u user] [-project project] [-public | -private]
      [-r] [names… | -a]

DeleteRevisions [-u user] [-project project] [-file] [-y]
      revision…

MergeBranch file

ModifyReadOnly file…

MountProject

NameRevisions [-u user] [-project project] [-public | -private |
      -b][-r] [[-only] | name [[-expand] [-s] | [-replace]
      [-dynamic] [names… | -a]]]

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

OrphanFiles file…

Project [-q | projectname]

ProjectInfo [-project project] [-log] [-comments] [-latest] [-f]
      [-r] [-s] [-only | -m] [-af author | -a author]
      [-df dates | -d dates] [-cf pattern | -c pattern]
      [-t pattern] [-n name] [-update | -newer] [path…]

TransferCkid sourcefile destinationfile

UnmountProject -a | project…
```