RUNNING SCRIPTS

INTRODUCTION

This chapter explains Resorcerer's simple text script language, which lets you build a new resource file from many existing ones all in one step. Resorcerer scripts implement a small subset of Apple's Rez language, with some minor enhancements and syntax extensions.

Resorcerer's scripting mechanism is *not* at this time a complete resource compiler like Rez/SARez. However, it does have enough capabilities to do useful automatic resource file management sorts of things. Resorcerer is itself now being built with the aid of its own scripting mechanism. For instance, we manage the building of both a Demo version and the shipping version of Resorcerer from a single script file that works in conjunction with our development system's C header files.

TOPICS COVERED

- Creating a script
- Running a script
- Preprocessor directives
- Script language syntax
- Keyword statements
- Examples
- Errors

CREATING A SCRIPT

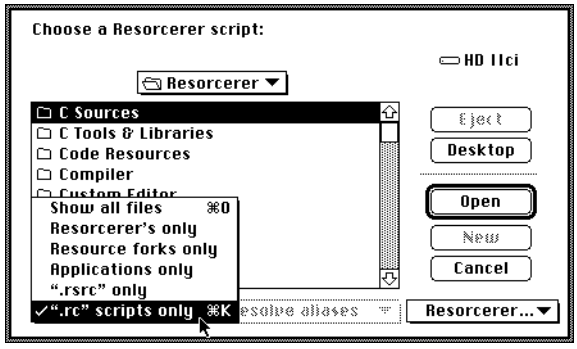
A Resorcerer script is a file of type 'TEXT' whose name ends in the current script suffix. When Resorcerer is shipped, this suffix is set to a default of ".rc". The default suffix is ".rc" rather than ".r" to help distinguish Resorcerer scripts from Rez source files. However, you can change the suffix to anything you want.

To set the default script file suffix, use the **File Preferences** section of the **Preferences...** command in the **Edit** menu.

The script is kept as a stream of standard ASCII text in the text file's data fork, and you should be able to create the script using your favorite source code editor the same way you would create any other text source file.

RUNNING A SCRIPT

Resorcerer's script mechanism is only available when running under System 7.0 or later. From the Finder, you can run a script by dragging and dropping its file icon onto Resorcerer's icon.



Within Resorcerer, the usual **Open File...** command lets you choose from a variety a file types using a popup menu below the file/folder list. The last popup entry sets the list to show you only script text files, as defined by the current script suffix. When this type of file filtering is on, opening a script is a request to Resorcerer to *run* the script rather than to edit any resources that might be in the script file's resource fork. If you need to edit actual resources in the script file's resource fork, you must choose one of the other file filtering modes to open and get at the script file's resources.

There is no extra menu command in Resorcerer's **File** menu for directly opening a script. You have to choose **Open File...** and set the file type popup to ".rc scripts only" (see the following Sorcery, however).

Sorcery: When the **Open File...** dialog is showing, ⌘O and ⌘K are popup menu shortcuts that let you alternate between displaying all files and showing scripts only. ⌘K, like ⌘O, also brings up the **Open File...** dialog if it isn't already showing, but it presets the file filtering popup to display scripts only.

Sorcery: To suppress the display of folders, hold the Option key down. Volumes, although logically folders, cannot be suppressed. This shortcut may not work if you have third-party Standard File extensions installed.

Sorcery: Whichever of the various resource file editing filters in the popup menu has ⌘O attached to it is taken as the type of filtering to use the first time you run Resorcerer. If you want to set the default to something else, like "'.rsrc' only", use Resorcerer to edit a copy of itself, search for the popup 'MENU' resource, and move the ⌘O to the menu item representing your preferred filtering state.

FAVORITE SCRIPT FILES

You can include a script file in your list of favorite files attached to the "Quick Open" sub-menu. When you edit the Favorite Files list in the **Preferences** dialog, the state of the file type popup is kept with the file entry. Remember to set the popup to display scripts first before clicking the **Add** button. Otherwise the script file will be quick-opened as a normal resource file would instead of being run. In the event that there is a conflict, just delete the file from the list and add it again using a differentfile setting.

SCRIPT INTERPRETATION

When you run a script, Resorcerer creates a hidden, in-memory new resource file, called the working file, whose name is the same as the script file's, with the exception that the current script suffix is replaced with the current default resource file suffix. For the standard defaults, this means that, for example, running the script "Foobar.rc" will cause Resorcerer to create the (hidden) working file, "Foobar.rsrc". The Finder type and creator codes automatically default to the same as those Resorcerer uses for other new resource files.

If, during interpretation, the script explicitly specifies an output file

name, then Resorcerer renames the working file to the new output name, and will eventually save the working file to disk when the script finishes successfully. Otherwise (that is, if the script does not specify an output file) the working file maintains its default name but is simply opened as a new file for interactive editing when the script ends. This means that the completely empty script file (one with no commands in it at all) opens a new empty resource file for editing (this may change in future versions).

Resorcerer puts up a progress dialog to indicate what it is up to as it executes the script. Because scripts can include both symbolic headers and resource files, it is difficult to know beforehand how long a given script is going to take to execute. When resources are included from other files, execution slows down quite a bit due to the Resource Manager's disk accesses, but it is possible to tell beforehand how much work is about to be performed, so in these cases a progress bar is updated in order to keep you from going to sleep. If you are having a nightmare anyway, you can still hit ⌘ . (period) to stop the running script early.

Should an error occur while running the script (usually a syntax error, a file-not-openable error, or an out-of-memory error), it is reported in an alert and execution of the rest of the script is aborted, the working file in its current state is made interactively editable, and an attempt is made to close it. You can confirm or not, depending on whether or not you want to check anything or otherwise muck about in the new file. Generally, you will want to click on the **Don't Save** button.

For more on errors, see the later “Errors” section in this chapter.

PREPROCESSOR DIRECTIVES

Resorcerer script files are composed of a simple declarative language. In addition, there is a preprocessor for implementing simple symbolic macro substitution and conditional interpretation.

Preprocessor directives are currently a subset of the Rez and C preprocessor directives. The proprocessor is not a full ANSI C style preprocessor.

The preprocessor supports the usual C syntax of

```
#directive arg
```

The # must be the first non-white space character on a line. The preprocessor does not support any expression evaluation at this time. Its primary purpose is to be able to read reasonably simple C header files in which you keep only your symbolic resource ID definitions or enumerations.

The following directives are supported:

```
#include      filepathname
#define             macroname macrotext
#undefine     macroname
#undef        macroname
#ifdef        macroname
#ifndef             macroname
#else
#endif
```

In addition, there is a script statement for C-style enums, which are converted into sequential symbolic definitions of numbers in the preprocessor dictionary (see below).

INCLUDING OTHER HEADER OR SCRIPT FILES

```
#include filepathname
```

This suspends input from the current script text file, and begins reading input from the given file. The file must have type 'TEXT' or an error occurs. The pathname must be contained in standard string (non-typographic) double quote marks, as in

```
#include "::Headers:MyApplication.rc.h"
```

Included text files can themselves contain nested #include directives, up to some maximum inclusion level. You have to be explicit about location of the target file with respect to the script file's own folder. Resorcerer doesn't search all subdirectories for the first name match.

Under System 7, if the included *filepathname* is an alias file, then Resorcerer will resolve the alias and open the target file for inclusion. It only does this if the "Resolve alias files" setting is in effect, as shown by the popup in the **Open File...** dialog.

The most common use for this directive is so that you can create a header file in which you keep symbolic definitions for the resource IDs that your application passes to the Resource Manager subroutines. This ID definition file can then be used by your C compiler, Rez, or Resorcerer to specify resources by symbolic (i.e. semi-readable) name, rather than obscure number.

MAKING SYMBOLIC SUBSTITUTIONS

```
#define macroname macrotext
```

This enters *macroname* into the preprocessor's dictionary of symbolic macro names with a definition of *macrotext*. *macroname* must start with an alphabetic character or an underscore, and can be composed of up to 255 alphanumeric/underscore characters. *macrotext* is everything past *macroname* up to the first un-escaped carriage return. During subsequent scanning of the input script text, any identifiers that match the defined *macroname* will have *macrotext* substituted for them on the input, whereupon the defining text is re-scanned with possible other substitutions.

These macros cannot take parenthesized argument lists. Also, the case of the characters used in macroname is ignored. "OpenDialogID" and "opendialogid" are recognized as the same name.

It is an error to attempt to define a name that is already defined in the dictionary.

The most common use of the #define statement is to define names for the purposes of conditional operations, or to create symbolic names for resource IDs. A file containing nothing but resource ID definitions can then be included as a header for your application's C compiler, Rez compiler, or Resorcerer script using the #include directive.

```
#undefine macroname
#undef macroname
```

Either of these (they both do the same thing) removes a previously defined *macroname* from the dictionary. It is an error for *macroname* not to be found in the dictionary.

CONDITIONAL SCRIPT INTERPRETATION

```
#ifdef macroname
...
#else
...
#endif
```

This creates a conditional section of your script, which is scanned and either executed or ignored depending on whether the given *macroname* has been entered into the preprocessor's dictionary or not. If it has, then script text up to the next matching `#else` or `#endif` is included. If not, then script text between the next matching `#else` and its matching `#endif` is included. `#ifdef/#else/#endif` directives can be nested up to some maximum level. The `#else` clause is optional.

The most common use for these is so that you can write just one script file that conditionally does different things depending on a single defined name. This typically happens when you have more than one version of an application (e.g. a working and a demo version), where the two share the majority of your project's resources, but there are still some minor differences in resources that both versions contain.

For instance, Resorcerer's Demo Version includes all the various TMPLs directly in its own resources; whereas the shipping version keeps them in their respective files. So our script for building Resorcerer uses an `#ifdef` to optionally include them depending on whether a constant, `_DemoVersion_` is defined in our `CompilerFlags.h` file.

```
#ifndef macroname
...
#else
...
#endif
```

This is the same as the `#ifdef` directive described above, except that the sense of the condition is reversed. Here script commands between the `#ifndef` and its `#else` are executed if *macroname* is not defined. The `#else` clause is optional.

SCRIPT LANGUAGE SYNTAX

Once the preprocessor has done its secret work of substituting macro definitions for names, Resorcerer scans the input for white-space-separated tokens (numbers, identifiers, operators, etc.). It also ignores any comments.

COMMENTS

Comments have the form

```
/* This begins a comment.  In Resorcerer scripts,
   comments can nest, so that all of these following
   comments are processed correctly....

/* This is a standard C- and Rez-style comment */
/• This is a variant comment (bullets are Option-8) •/

 •   Everything to the end of this line is ignored
 //  So is everything to the end of this line

 ... and we end our original comment here */
```

TOKENS

Tokens are of the form:

```
     alphanumeric   kFoobar         _undershirt_   AB4039  enum
     decimalnum     128    -41    +7
     hexnum         0xFFFF    $80
     literal        'C' 'DLOG' ‘snd ’ 'Tab\t'
'A\$42\0x43\0d064'
     string         "abcdefghijklmnopqrstuvwxyzebraandbeyond"
                    “Error\r”  "\""  "\0b10101010"
     operators      =   (    {   +   ,   ;  <<  !=
```

Strings and literals can also be enclosed in typographic open and close quotes.

Within literals and strings, the usual C and Rez backslash escape sequences for non-printable tabs `'\t'`, backspace `'\b'`, return `'\r'`, newline `'/n'`, form feed `'\f'`, vertical tab `'\v'`, and rubout `'\?'` characters are supported. To include a backslash, or quotation mark of the same type as you are using to delimit the string or literal, precede it with a backslash. You can also use 8-digit binary, 3-digit octal, 3-digit

decimal or 2-digit hex escape sequences (see the examples above).

EXPRESSIONS

In the places were a script statement accepts numeric arguments, the interpreter will perform standard expression evaluation, using Rez (or C) operators, in order of precedence:

`( expr )`	Parentheses to force precedence
`- expr`	Unary minus (twos complement)
`~ expr`	Bitwise negation (ones complement)
`! expr`	Logical negation (0 or 1 from 0 or not 0)
`expr1 * expr2`	Multiplication
`expr1 / expr2`	Division
`expr1 % expr2`	Remainder
`expr1 + expr2`	Addition
`expr1 - expr2`	Subtraction
`expr1 << expr2`	Left shift by `expr2` bits
`expr1 >> expr2`	Arithmetic right shift by `expr2` bits
`expr1 > expr2`	Greater than
`expr1 < expr2`	Less than
`expr1 >= expr2`	Greater than or equal
`expr1 <= expr2`	Less than or equal
`expr1 == expr2`	Equal
`expr1 != expr2`	Not equal
`expr1 & expr2`	Bitwise AND
`expr1 ^ expr2`	Bitwise XOR
`expr1 | expr2`	Bitwise OR
`expr1 && expr2`	Logical AND
`expr1 || expr2`	Logical OR

The logical operators &&, ||, <, <=, ==, !=, >=, and > evaluate to 1 or 0 (true or false).

Note: Rez-style $$ string variables are not supported.

ENUMERATED RESOURCE IDS

Enumerations are nice ways to create long lists of symbolic resource IDs while guaranteeing that each symbolic name has been defined to a unique ID, and they are used quite a bit in C program header files. enum's are most useful for multiple resources of the same type.

The syntax of the enum statement is:

enum { *name [= number] , ... , name [= number]* } ;

Each *name* is defined in the preprocessor's dictionary to a number (actually, a string of digits). If a *name* is followed by an optional = *number*, then the digits of *number* are used; otherwise, the number entered is 1 more than the previous number defined in this enum. If the first *name* doesn't explicitly provide a *number* to start the enumeration count, then the count is started at 0.

An example of an *enum* declaration might be something like...

```
enum {
     // start enumeration at 128 instead of the usual 0
     kProgressDialogID = 128,
     kSearchDialogID,          // implicitly defined as
"129"
     kCompareDialogID = 140,   // explicitly equals "140"
     kPreferencesDialogID,           // implicitly "141"
     kSaveChangesID                  // "142"
     };
```

KEYWORD STATEMENTS

Script statements have the following general form:

keyword (*arglist*) { *subdeclist* } ;

The *arglist* and its parentheses, and the brace-enclosed *subdeclist* are all optional. *keywords* are case-insensitive.

SYNOPSIS

Resorcerer's script interpreter supports the following statements:

```
create filepathname ;
filetype ( literaltype ) ;
creator ( literaltype ) ;
include filepathname [ rangelist ] [ except rangelist ] ;
read resource [ from ] filepathname [ offset [ : size ] ] ;
delete rangelist [ except rangelist ] ;
change attributes [of] rangelist [ except rangelist ]
                                        [to] attributelist ;
change ID [of] rangelist [ except rangelist ] [to] num ;
change name [of] rangelist [ except rangelist ] [to]
namestring ;
change type [of] rangelist [ except rangelist ] [to]
literaltype ;
```

RESOURCE RANGE SPECIFICATIONS

Most of the script statements take as arguments either a single resource specification, *resource*; a resource range specification, *range*; a resource range list, *rangelist*; or a pair of *rangelist*s separated by the except keyword, where the second *rangelist* specifies sets of resources to exclude from the first *rangelist*.

A single *resource* specification consists of a four-character literal resource type and a numeric resource ID expression. If you don't care about the resource attribute bits or optional resource name, you can simply write the type and ID. If you want to specify a name or attributes, they, along with the ID, must be enclosed in parentheses.

Examples of legal single resource specifications include:

```
#define kPreferencesBaseID 500

'DLOG' kPreferencesDialogID + 2
'TEXT' (129,"Preamble")
'CODE' (2,preload,locked)
'STR#' (preload, 41*7, "Errors", purgeable)
```

Note: The Rez language often requires the ID to be parenthesized even when the other optional stuff is not necessary, and may further require the ID, name, and attributes to occur in that order.

Note: Regardless of Resorcerer's **Preferences...** setting for including related or owned resources in interactive selections, script files only operate on *explicitly* described resources or ranges of resources.

By default, the resource name is the empty string, and the attribute bits are all cleared. The parenthesized list of items can contain, in any order, one resource ID integer, zero or one resource name string in quotation marks, and any of the standard Rez-style attribute bit names:

clear bits			*set bits*
	appheap	sysheap	
	nonpurgeable	purgeable	
	unlocked	locked	
	unprotected	protected	
	nonpreload	preload	

A resource *range* consists of a 4-character resource type, optionally followed by an ID or ID range and/or other optional parenthesized info that defines which resources fit in the range. An ID range is either a single numeric expression or a pair or expressions separated by a colon.

Examples of legal resource *range*s:

```
'STR#'                  // All resources of type 'STR#'
'STR#' 100:200          // All 'STR#'s with ID's in the
                        // range 100 to 200, inclusive
'STR#' ()               // All resources of type 'STR#'
'STR#' (128:32767)      // All 'STR#'s with ID's greater
than
                        // or equal to 128
'STR#' ("")             // All 'STR#'s with no name
'STR#' ((1<<7),"")      // 'STR# 128, but only if it is
unnamed
'STR#' (-32768:-1,"Errors")  // All 'STR#'s with negative
IDs
                             // and named "Errors"
```

A resource *rangelist* is a comma-separated list of 0 or more resource *range*s. If the list is empty, this is usually taken to mean all resources in the file in question.

An example of a single legal resource *rangelist* with 8 resource ranges in it might be:

```
'STR#', 'TEXT', 'STR ' 128:400,
'AUTO' 128, 'SIZE' (-1:0), 'XCMD' ("LegalFoo"),
'TMPL' (0:32767), 'FLTR' (0:32767);
```

SPECIFYING THE SCRIPT'S OUTPUT FILE

`create` *filepathname*

The `create` keyword declares the name of the disk file to which Resorcerer should save the current working file when the script ends. This keyword has no immediate effect other than to rename the current internal working file, which Resorcerer automatically creates in memory when you first run any script, but which is not written to disk until later on when the script ends.

The *filepathname* should be a quoted string.

If the script has no "create" keyword in it, the current working file will be opened for editing when the script finishes.

Note: This keyword is not Rez-compatible.

Examples:

```
create "New Demo.π.rsrc"
Create "MyApplication.rsrc";
create("NoSuffix Resources")
```

`filetype (` *literaltype* `)`
`creator (` *literaltype* `)`

The working file is given the same Finder signature type and creator codes that Resorcerer normally gives any new file (which are, respectively, `filetype('RSRC')` `creator('Doug')`). You can override either of these explicitly using the `filetype` or `creator` keywords.

Examples:

```
filetype('UNIV') Creator('G•D!')
```

```
FileType('PLCY') creator('WONK')
```

Caution: If the file specified in your `create` statement already exists on your disk, it will be overwritten without the slightest hesitation and all previous contents will be lost.

INCLUDING RESOURCES FROM OTHER RESOURCE FILES

`include` *`pathname`* `inputrangelist` *`[`* `except` *`skiprangelist ]`* `;`

The `include` statement opens a given file of resources and reads into the working file all resources in *`pathname`* that belong to any range in *`inputrangelist`* but that do not belong to any range in *`skiprangelist`*. The `except` clause is optional, and for compatibility with Rez you can use `not` in place of `except`.

Under System 7, if *`pathname`* is an alias file, then Resorcerer will resolve the alias and open the eventual target file for inclusion. It only does this if the **Resolve alias files** setting is in effect, as shown by the popup in the **Open File...** dialog.

Compatibility note: Other Rez compilers may not support an *`inputrangelist`* or *`skiprangelist`*, only an *`inputrange`* and *`skiprange`*.

The `except` keyword keeps resources falling into its *`skiprangelist`* from even being loaded, so it is useful, for instance, when you need to exclude a very large resource for which there is not enough memory. Using the `delete` keyword, described later in this chapter generally requires resource data to be already loaded into the working file.

It is currently not illegal to include resources with IDs that duplicate already included resources. However, when the script ends the working file is checked in the same manner as all newly opened resource files are, and any duplicates will be reported to you.

Examples of including resources with various types of ranges might be:

```
include "Original Preferences.rsrc";
```

which adds all resources from the file `Original Preferences.rsrc` to the working file.

```
     include "MasterTemplates.rsrc" 'TMPL', 'FLTR'
       except 'TMPL' ("PRV#'), 'FLTR' ("C   PRV#"), 'TMPL'
("");
```

adds all 'TMPL' and all 'FLTR' resources from the file `MasterTemplates.rsrc` except any 'TMPL' named "PRV#", any 'FLTR' named "C PRV#", and all 'TMPL's with empty names.

```
     include "Sound Library alias" except 'snd ' 777,
                                        'snd ' 1000:2000 ;
```

adds all resources in the file `Sound Library` to which `Sound Library alias` points, except 'snd ' 777 and all sounds with resource IDs between 1000 and 2000, inclusive. The *`inputrangelist`* in this case is the empty list, which is taken to mean all resources in the file.

CREATING RESOURCES FROM A FILE'S DATA FORK

`read` *`resource`* `[from]` *`filepathname [ offset [ : size ]  ]`* `;`

The `read` statement lets you read a given file's data fork, or any part of the data fork, as a single resource. This is useful, for instance, when you want to create large 'TEXT' resources for which a better text processor program exists than Resorcerer's TextEdit-based Text Editor. The ability to specify the range of bytes in the data fork to read lets you pull in, for example, 'PICT' data files as resources of type 'PICT', since the data in a 'PICT' file contains a 512 byte header that the resource won't have.

If *`offset`* is missing, all data from the file is read into the resource. If *`offset`* is present but *`size`* is missing, all data from the given offset to the end of the file is read into the resource. If both are present, separated by a colon, a block of size bytes is read from the given offset.

Under System 7, if *`filepathname`* is an alias file, then Resorcerer will resolve the alias and open the target file for inclusion. It only does this if the **Resolve alias files** setting is in effect, as shown by the popup in the **Open File...** dialog.

The optional `from` preposition is included for readability and may not be Rez compatible. The ability to read partial data forks may not be possible either.

Examples of legal data fork read statements might be:

```
read 'TEXT' (128, "About Box", purgeable)
                              from "About Box Text";
```

reads all the text in the file `About Box Text` into a new 'TEXT' 128 resource with the given name and attributes.

```
read 'PICT' 128 from "::Images:MacPaint Idea" 512 ;
```

creates a new 'PICT' 128 with all data except the first 512 bytes of the file `MacPaint Ideas`.

```
enum { kAppIconID=128, kDocIconID };
read 'icl8' kDocIconID from "Icons.data" 0x1000:$400;
```

reads 1024 ($400) bytes, starting at offset 0x1000 from the data in file `Icons.data` and places them in an 'icl8' resource with resource ID 129.

DELETING RESOURCES FROM THE WORK FILE

`delete` *`rangelist [`* `except` *`skiprangelist ]`* `;`

The `delete` statement removes all resources in the current working file that are included in the given *`rangelist`*, but not included in the given *`skiprangelist`*. The `except` clause is optional, and you can use `not` for Rez compatibility. If the *`rangelist`* is empty and there is no `except` clause, then all resources in the current workfile are cleared.

Note: Some Rez compilers may not support the deletion of either the empty *`rangelist`* or anything other than a single *`range`*, and may not support the `except` (or `not`) clause either

Examples of legal `delete` statements:

```
delete ;
```

deletes all resources in the current in-memory working file.

```
delete 'DLOG', 'DITL' 200:299, 'ictb' 200:299, 'dctb';
```

clears out all 'DLOG' and 'dctb' resources, and all 'DITL' and 'ictb' resources with IDs in the range 200 to 299, inclusive.

```
Delete 'DLOG' 128 ;
```

deletes 'DLOG' 128 but does not delete anything else, such as 'DITL' 128. Note again that upper/lower case in the keyword makes no difference.

```
Delete 'SHIP' except 'SHIP' 128:140 ;
```

deletes all 'SHIP's except for 'SHIP's 128 through 140, inclusive. No other related resources are deleted.

```
delete 'MENU' ("File"), 'MENU'("Edit");
```

deletes all menus with resource name "File" or "Edit" (presumably there will be only 1 of each).

Remember that it is much better to exclude a resource from being included using the `include … except …` statement than it is to include it and then delete it, since in the former case, no memory is required to load the resource data into the work file. The `delete` statement will be more useful when opening and editing existing files by script, which is not yet supported.

CHANGING RESOURCE NAMES

To rename any range of resources, excluding another range, in the current work file, use:

```
change name [of] rangelist [ except skiprangelist ]
                                          [to] namestring
;
```

This renames all resources in *rangelist* that are not also included in *skiprangelist*. *rangelist* can be empty, which is taken to mean all resources in the current working file. The name of each affected resource is changed to *namestring*. The most common use for this is to strip the names from any or all resources in a file.

Stripping names (we like to call it "name-dropping") is often applied before creating the final resources for an application about to be shipped. Doing so can save some disk space, and makes it less pleasant

for others to muck about in your application's resources with their favorite resource editor than it otherwise would be. Of course, you should only strip resource names that are there solely to make editing them easier; names that your application depends on to differentiate among resources of the same type should not be stripped (of their individuality and dignity, no less!).

Another use of the rename command might be to rename all the 'CODE' segments in a built application to something reasonable. Some compilers don't provide a way to give names to the 'CODE' resources they create. Named code resources make MacsBug or other debugger output somewhat more understandable during Heap Dumps. Once the script is written, you can run it easily after each application build to re-install your favorite names.

Examples of legal namedropping:

```
change name to "";
```

removes the names from every resource in the current working file.

```
CHANGE Name of 'DLOG', 'ALRT','DITL','STR#';
```

strips the names from every 'DLOG', 'ALRT', 'DITL', and 'STR#' in the current working file. Remember that the interpreter ignores case except in literals and strings, which is why there is no difference between "change" and "CHANGE" in the above.

```
change name except 'TMPL', 'SHOW', 'FLTR',
              'MPTR' 128, 'TYP#' table1_ID:table4_ID to
“”;
```

removes the names of all resources in the working file except 'TMPL's, 'SHOW's, 'FLTR's, 'MPTR' 128 only, and all 'TYP#'s with IDs in the given symbolic range. You don't want to remove the names of certain resource types, such as 'TMPL's in Resorcerer, or 'XCMD's in HyperCard stacks, because their names contain information that the application depends on.

```
change name of 'CODE' 0 to "Jump Table";
change name of 'CODE' 1 to "SetUpA5";
change name of 'CODE' 2 to "Main";
```

```
change name of 'CODE' 3 to "Library routines";
...
```

might be what a possible script would look like for re-installing 'CODE' resource names after a new compiler build has created unnamed 'CODE' resources.

Note: Newer versions of your development system, such as THINK C™, may have had the ability to name code segments added.

CHANGING RESOURCE ATTRIBUTES

```
change attributes [of] rangelist
            [ except skiprangelist ] [to] attributelist
;
```

lets you set attribute bits in all resources in *rangelist* except any in *skiprangelist* using the attributes specified in *attributelist*. The of and to keywords are optional.

attributelist can be either a numerical expression or a list of comma-separated attribute names:

sysheap, purgeable, locked, protected, preload
appheap, nonpurgeable, unlocked, unprotected, nonpreload

For example,

```
change attributes to purgeable;
```

ensures that every resource has its purgeable bit set without touching any other attributes.

```
change attributes to nonpurgeable;
```

removes the purgeable attribute from every resource in the working file.

```
change attributes of 'DLOG', 'ALRT', 'DITL'
       except 'ALRT' 128:129, 'DITL' 128:129
       to purgeable, nonpreload;
change attributes of 'ALRT' 128:129, 'DITL' 128:129
       to nonpurgeable, preload;
```

ensures that every 'DLOG', 'ALRT', and 'DITL' is purgeable except for one particular pair of alerts (128 and 129), which should always be

loaded and available. Any of the other attributes are left unaffected.

If you want to guarantee that all attributes of a resource are exactly what you want, you have to explicitly declare how you want each bit set using a list of five attribute keywords. Or you can explicitly declare the value of the attributes using an expression:

```
     #define resProtected       0x08
     #define resLocked   0x10
     change attributes of 'CODE' 1:128 to
resProtected|resLocked;
```

CHANGING RESOURCE IDS

change ID *[*of*] rangelist [* except *skiprangelist] [*to*] expr* ;

lets you assign a new resource ID to all resources in *rangelist* that are not also in *skiprangelist* . The new ID is the value of *expr*.

It is an error for there to be more than one resource in *rangelist* having the same type.

Example:

```
     change ID of 'ZTAB' 128 to (kNewBaseID+128);
```

Note: Unlike Resorcerer's interactive **Change All...** command (see the **Editing Resources** chapter for more on this), the change ID script statement does *not* renumber ranges of resources to sequences of resource IDs starting at *expr*. It also does not renumber any related or owned resources, nor does it affect any internal resource data that might have to be renumbered to maintain consistency among resources in a set.

CHANGING RESOURCE TYPES

change type *[*of*] rangelist [* except *skiprangelist] [*to*] type* ;

The change type statement changes all resources in *rangelist* in the working file that are not also in *skiprangelist* to the new

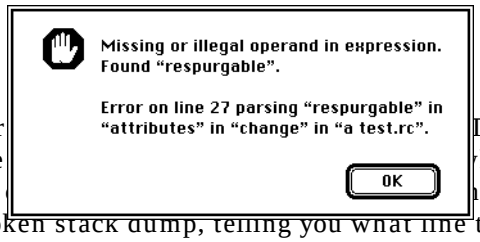
resource *type*, where *type* is a 4-character literal.

The interpreter will complain if any single `change type` statement would create a duplicated resource ID in the current working file.

Examples:

```
     change type of ‘STR ’ 129 to ‘Estr’;  • Change one
resource
     change type of ‘STR#’ to ‘str#’;    • Change all ‘STR#’s
     change type to ‘GLOB’;              • Change all
resources
```

ERRORS



The script interpreter reports error[illegible]. The general format of an error message[illegible]what was expected, and what was found[illegible]his message is usually followed by a token stack dump, telling you what line the problem was found on, and the nested context of the error.

For instance, the following script statement misspells `resPurgeable`:

```
#define resPurgeable (1<<5)
change attributes of 'SPUD' 77 to resPurgable
```

which results in the alert: