Chapter 4 - User Interface

This chapter describes the user interface, how the project structure works, and what all the menu items do.

Project structure

files

When ReWrite is first opened, it prompts for a file to be opened (called a project file hereafter). This should be a text file listing all the files containing ReWrite code to be used.

Note: ReWrite is not 'drag and drop'. In addition, the application should be opened by opening the application itself, not any ReWrite files.

For each name in the this file, the name preceded with '`r.`' contains the source code, and the name preceded with '`l.`'. These files must be in the same directory as the project file.
For example, the sample project file is called `0Project`, and has the following contents:

```
setup
go
```

This indicates that there are two files or ReWrite source code, called `r.setup` and `r.go`, and the object code will be called `l.setup` and `l.go`.

If any of the files do not exist, they will be created when needed.

The `r.` (source) files can all be opened from the Windows menu. It is most important to open source files from the Windows menu rather than using the Open… in the File menu. The File menu way treats the file as just another document, so any changes that have been made will not be recognised by the compiler until the file is saved. Even worse, once changes are recognised as having been made, then the compiler will try to open the file again. This could cause major problems.

build order

When a rule definition is being looked for, first the internal libraries will be checked, then the files in the project will be checked for from the bottom file up.
This contrasts with checking from within the file, so in the example above, the search will (after first checking the internal libraries) start at the top of `go` and finish at the bottom of `setup`.

changing project

If you wish to change a project without quitting ReWrite, select Change Project from the Project menu. This will do two things:

- it will close any open windows,

- it will prompt for a new project file.

If cancel is used anywhere during these, the Change project operation will be aborted and the original project will be left open. There must always be an open project within ReWrite.

Output

All output from ReWrite will go to the Output window. This is just an ordinary TextEdit window which can be edited. Output will be inserted wherever the cursor is in the window. If an operation would cause the output to exceed 32K, there will be a close dialog box brought up:

- If cancel is chose, the rest of the output will be lost.
- if the window is closed, a new blank one will be started when necessary.

Any time there is output but no Output window, a new one will be created.

The Output window can be selected from the first entry in the Windows menu.

Running ReWrite

All menus items listed in this section are in the Project menu.

There is 512K of memory allocated for ReWrite to use, and 64K of stack allocated. (These are configurable - see below).

build and go

Build updates the project. It:

- compiles any source files that have been opened and changed (remember to use the Windows menu to open files). Each changed source file is saved to disk just before it is compiled.
- if the source is not open, it compares the dates of the source and object code files, and opens and compiles the code if the object file is older.
- links all the code together so it is ready to run.
- display the names of any functions that were called but not found. (link errors)

As rules are complied, a list of function names will be output to tell you how far it s got. The compiler takes about one second per line, so be patient.

As each file is compiled, the address in memory that it has been compiled to will be displayed. This can be ignored.

Go runs the project. First it does a Build, and then calls the function `top[]` (with no arguments). The code will run from the first found match of this.

After the code has been run, the following will be returned:

- the amount of memory not used during any ReWrite code since the last time the memory was reset (see below). Note that this are not the amount of memory left - this will have been restored to its full value. This is given so that you can see when you are close to running out of memory.

- The time taken in ticks. (A tick is a sixtieth of a second).

aborting

The code can abort because of any of the following:

•
match error.
A rule couldn't be found that matched the arguments that were passed. The function name (or garbage if there was no rule found at all) will be displayed along with the offending argument.

•
user interrupt.
The user can interrupt any ReWrite code (including the compiler) by pressing âŒ˜-. (command-dot). This is taken from the OSEvent queue, and only the top event is checked, so if there has been another event, you may need to use âŒ˜-. up to 20 times before anything happens. Sometimes it takes a few seconds for anything to happen anyway, but I've never seen it fail.

•
stack overflow.
This is self explanatory.

•
memory overflow.
This is self explanatory.

•
division by zero.
This is self explanatory.

•
parse error.
This happens when the compiler encounters a syntax error, in which case the three lines of code preceding the error will be displayed, or a semantic error (unknown name on right, unknown type etc.) in which case the position of the error can be roughly worked out by looking at what function was last compiled.

Any of these will cause an appropriate message to be displayed in the output window, and then the memory reset if necessary.

Note that integer overflow is not checked for.

reset memory

In the current version, the ReWrite memory (the cells) only contain any useful information while a program is running. All the code, link information etc. is in fact stored on the Macintosh

heap. At the end of each piece of ReWrite code run, the freelist is checked to see if it is all there. If it is not, the entire ReWrite memory (freelist) is reinitialised. This should only happen automatically if there is an error that aborts the code. (If anyone can write a piece of ReWrite code that actually 'leaks memory', please let me know. I don't think that it is possible).

The reset memory option in the menu is provided to let the user reinitialise the ReWrite memory. The only use of this is that it resets the freelists, so that it can be found out how much memory a particular run of ReWrite code actually required (see above).

rulelist

Rulelist simply lists all the rulenames that the linker currently knows about. This is completely part of the user interface, so cannot be aborted part way through.

Limitations

There are several limitations imposed by memory and by the Pascal interface:

• TextEdit is used for the source and Output windows, so they are limited to 32000 characters. If an operation will cause the Output to grow bigger than this, the Output window will be closed (giving you the option of saving the contents), and a new empty one will be opened.

• There is a limit of 100 source files. This shouldn’t be a problem.

• There may be only 800 rule names in use at any one time (only 700 per file, although this should not be a problem). Going over this limit will cause a crash. The number of names left is given after linking. The library and compiler together use a total of about 300 names.

• Only the first 31 characters of any rule name are recognised, so function names must be distinct in the first 31 characters.

• Tokens have a limit of 250 characters.

Configuring ReWrite

There are several options that can be changed in ReWrite:

• The font
• The font size
• The number of cells
• The stack size

This can be adjusted using the 'Conf' resource ID = 128 in the application's resource fork. This resource is 12 bytes long and the format is:

```
word          font                default = 3 (Geneva)
```

```
word          font size          default = 9
long word     ReWrite memory     default = 512K
long word     stack size         default = 64K
```

Remember to change the size of the application to suit. (The application itself and libraries take up about 700K, and remember to leave space for your code). Doing this is most important, because the user interface does not check for 'out of memory' errors and will crash if one happens.

For anyone that is really interested, there are other ways of configuring ReWrite using resources, so that for instance all the libraries can be checked after the code, or all the libraries can be in external files rather than resources. If anyone really wants to play with this (it can make a mess if you make a mistake), E-mail me and I will tell you how to do it.

Future Compatibility

The following coding features that might (in my opinion) cause future compatibility problems, although at the moment everything seems to work on all the machines (including PowerMacs) that I have tried:

• `FlushInstructionCache` is used before any ReWrite code is run. If the data cache is not write-back then this should really be flushed too. At the moment, the operating system takes care of this, but it might not always. In reality, this shouldn't really be a problem anyway, because a lot of code is executed between the compiling and the running.

• VBL tasks are used for detecting the user abort. This seems to me like the sort of feature that future operating systems might limit.

• The 680x0 '`Trapcc`' and '`Divide by 0`' exception vectors are changed before any ReWrite code is run (so that any errors are detected), then changed back again afterwards. It is assumed that VBR = 0, as some computers seem to run in user mode and reading the VBR is a privileged instruction. I am not sure whether all future emulations of the 680x0 in the PowerMac will support this.