

EUDORA 6.X FOR MACINTOSH

BASIC PRODUCT ARCHITECTURE

Introduction

Eudora 6.x (and earlier) for the Macintosh (hereafter just “Eudora”) is a Carbon CFM application. It is largely object-oriented, and written in C. Much of the code predates the development of the Carbon framework, and traces of the original Macintosh Toolbox code are very much in evidence, including a `WaitNextEvent` loop instead of use of `CarbonEvents`. The codebase is old, and its complexity has increased with time. Much that is provided today “for free” to Cocoa applications had to be done “by hand” inside the Eudora application, increasing complexity still further.

The original implementation target included modestly-powered machines such as the Macintosh Plus, which supported an entire megabyte of memory. So much attention has been paid to efficiency and small footprint. While this attention was necessary at the time, it is largely irrelevant today, and occasionally actively harmful.

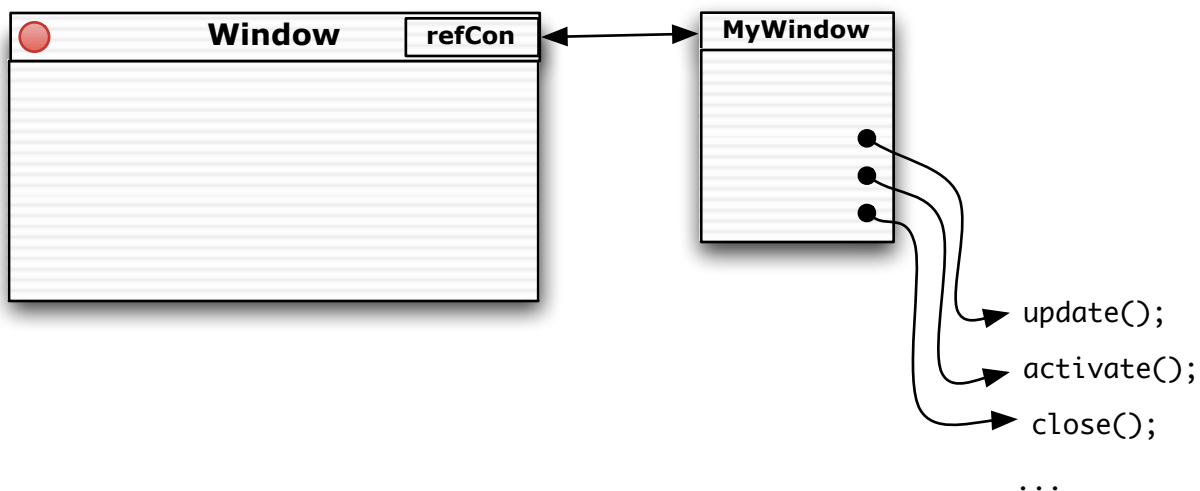
Building Blocks

Memory Management

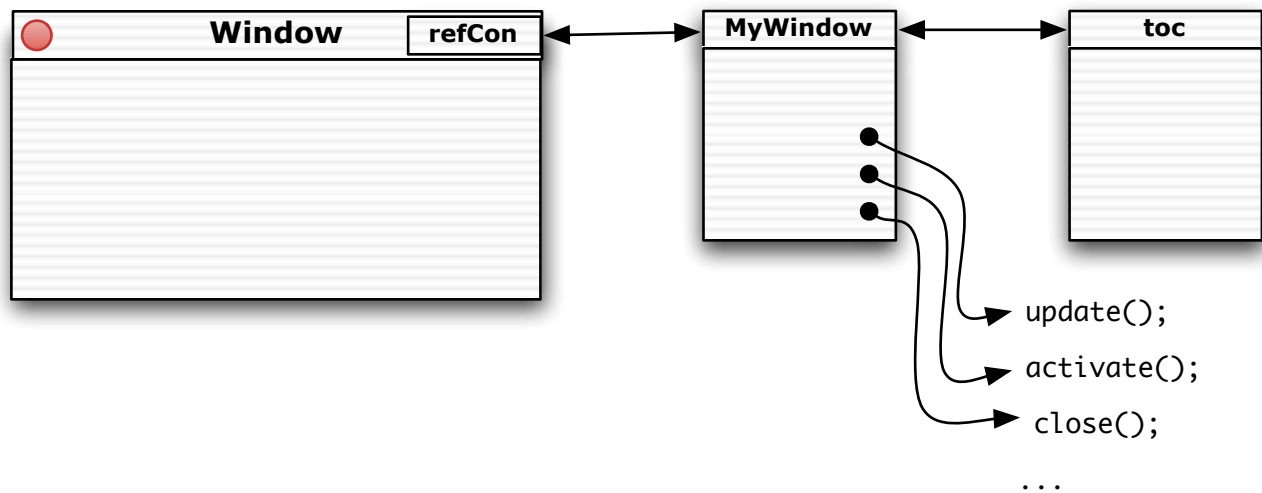
Under MacOSX, there is no reason to use the memory management scheme of the original Macintosh, with double indirect memory allocation. However, since Eudora was conceived nearly twenty years ago and was meant to be crammed into a very small amount of memory, it has always made full use of the relocatable memory management system. Handles are used throughout the application, and while it’s not necessary anymore to worry about them being relocated, it is necessary to follow the bouncing pointers with joy. Nearly all internal data structures are declared as struct’s, but actually used as handles to struct’s.

Windows

The most fundamental element in Eudora is the window. Very few objects are instantiated without a window being associated with them, even if the window is never shown on the screen. Events are often dispatched based on a table of function pointers associated with such windows. Originally, these windows were augmented toolbox windows, but now they consist of a `WindowRef` and an auxiliary structure hung off the window’s `refCon`.



Both Windows and MyWindows are used frequently inside Eudora, and often converted from one to the other. Furthermore, many other basic building blocks, such as messages and toc's, are also convertible to and from a window:



These conversions happen frequently, and the assumption that “there is always a window” is very deeply ingrained in Eudora.

TOC's

Central to message handling are Tables Of Contents, or toc's. They are used as the major object for a collection of messages, called a mailbox. A toc consists of a header that describes the mailbox as a whole (such as on-disk location, number of messages, name, etc) plus zero or more message summaries. (The structure itself contains one summary, but others are appended to the actual storage as need be.)

N
mymailbox
3 unread
...
summary 0
summary 1
summary 2
...
summary N-1

Message Summaries

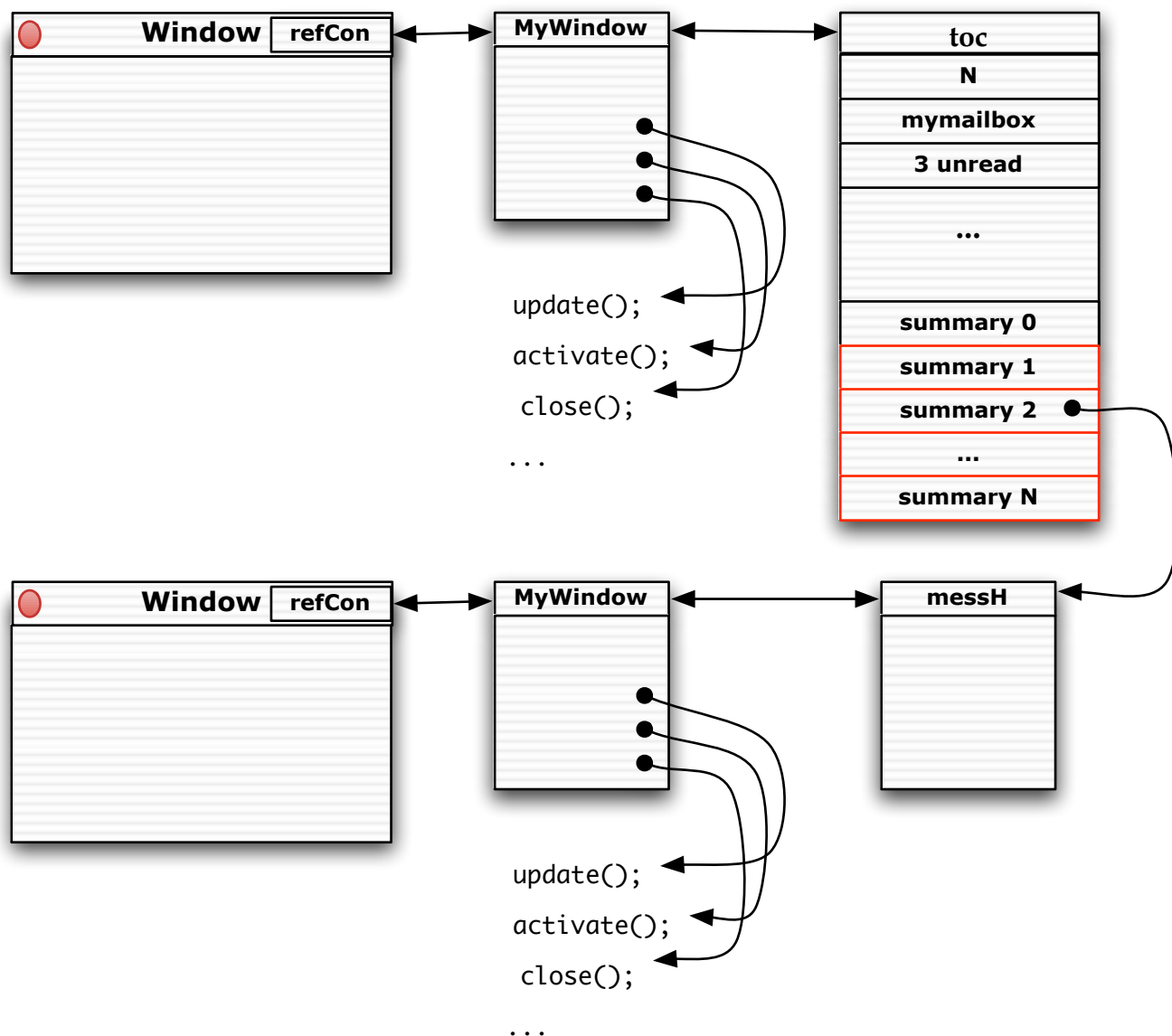
A message summary, usually called just a summary is a brief descriptor that allows a full message to be located and manipulated. The summary consists of both cached data (for example, the text of the subject and sender of the message), and metadata that is either computed (for example, whether or not the message contains attachments) or assigned (for example, the read/unread status of the message). One of the vital purposes of the summary is to locate the message inside the mailbox file on disk.

Summaries are written en masse to the disk when saving tables of contents. However, some items in the summary are in-memory only, and these are initialized to nil when the toc is read back from disk. One of

the most important of such in-memory items is the message cache, which is a cached copy of the entire text of the message. Often, this cached text can be used in lieu of actually instantiating the message object proper.

Messages

Messages are, needless to say, vital to the operation of an email program. In Eudora, a message is represented by a Window, with an associated MyWindow and mstruct contained in a handle (called a MessHandle). These are always pointed to from a summary in a table of contents, thus:

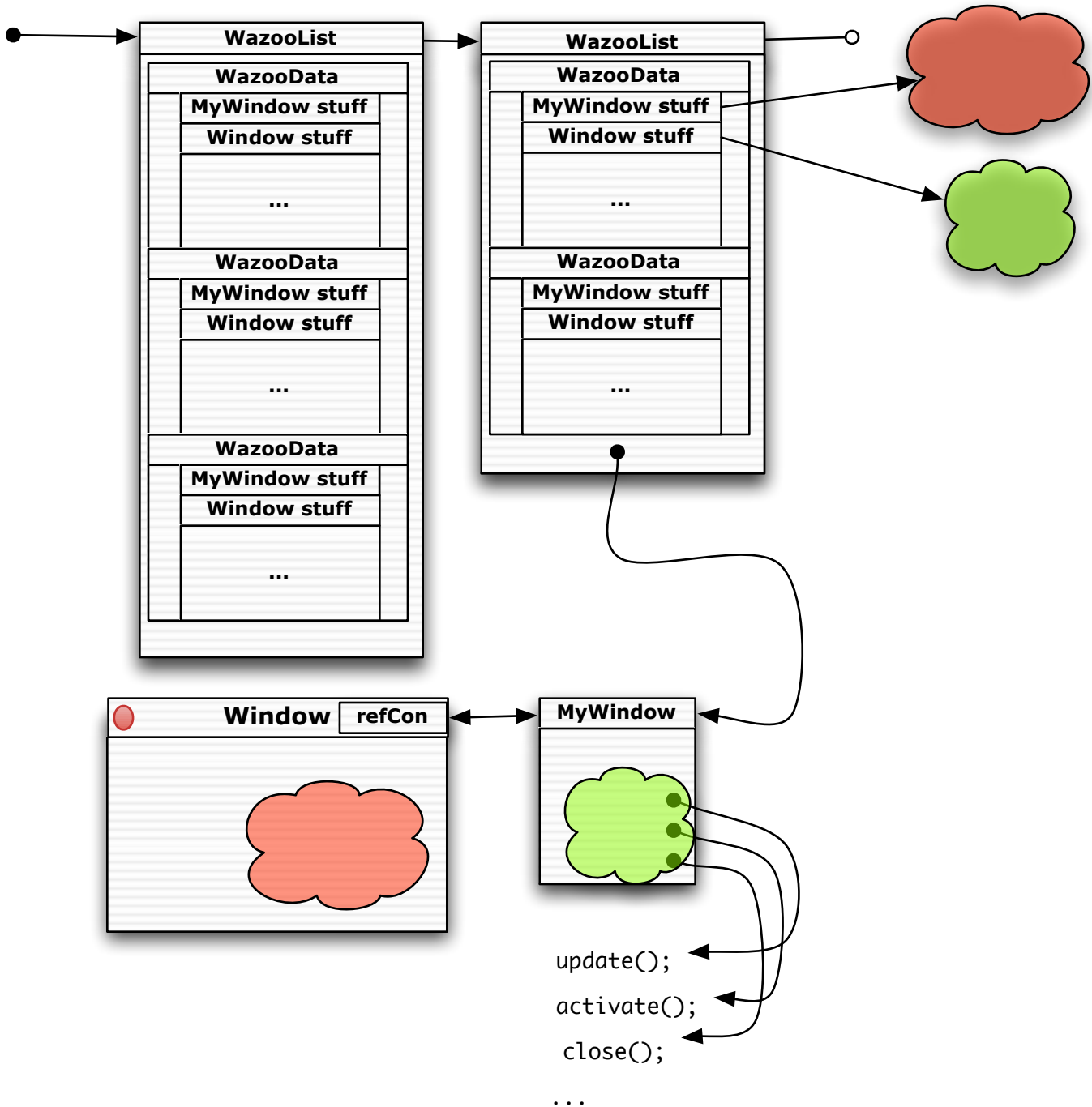


Composition Messages

Composition messages are messages the user has stored in the Out mailbox, and are either preparing to be sent or already sent. Internally, they are structured the same way as regular messages; the difference is that they have different action routines, not that the data structures differ.

Wazoos

Another important building block is the Wazoo. This is the internal name for what the user interface calls a “tabbed window”, and is the structure that allows the user to group utility windows together. Wazoos are implemented by swapping out select parts of the Window and MyWindow structures for a given window. Wazoos themselves are kept track of in a global list of WazooData structures. The whole ... collection looks like this:



When another tab is chosen, the noxious gassy areas of the the windows relevant to the current wazoo will be saved off, replaced with the noxious gassy areas of the new wazoo, and life will go on with the new wazoo taking control of the window. Note that only certain utility windows are wazoo-able; ordinary mailbox and message windows (currently) are not.

Event Processing

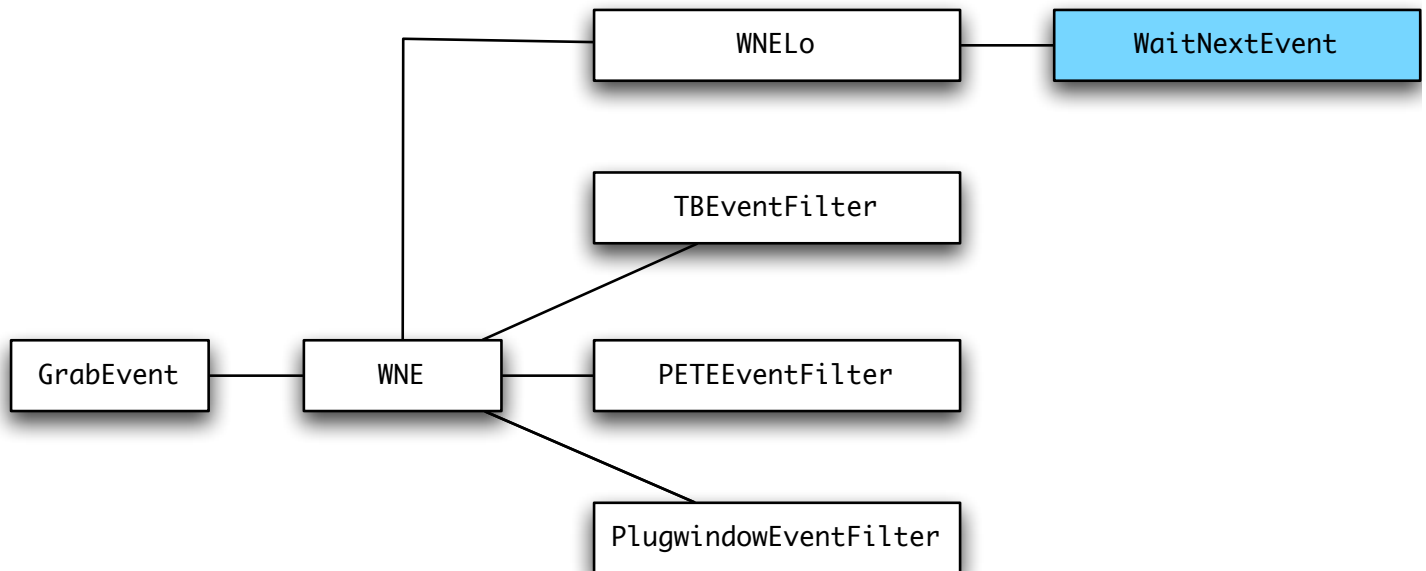
Eudora is a WaitNextEvent-based application. That means it has a central event loop that runs, calling WaitNextEvent to get the next event. Eudora also has other event loops sprinkled in the code, including one that polls for i/o and another that deals with moveable-modal dialogs.

But the vast majority of the event processing happens in the main loop, and that's what we'll concern

ourselves with here.

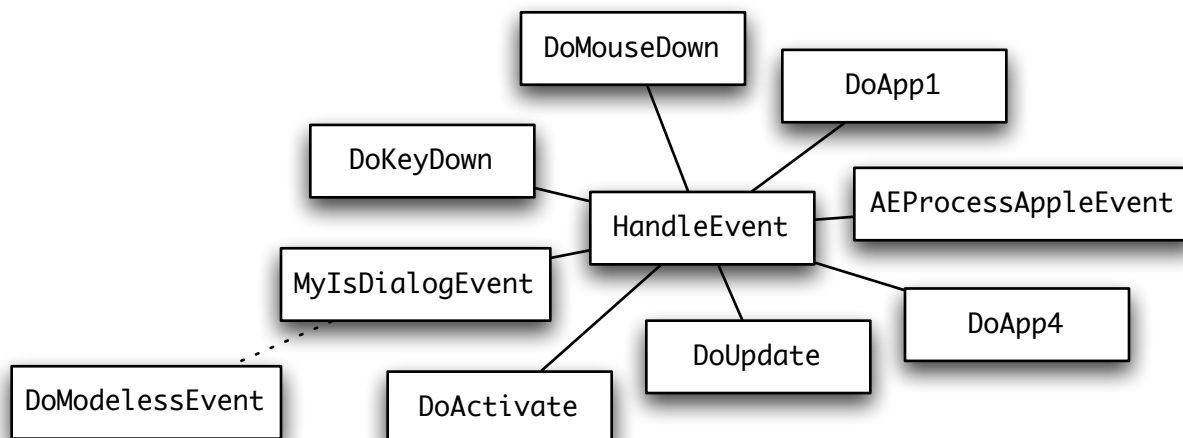
At the most basic level, the main loop calls “GrabEvent” to get an event, and “HandleEvent” to handle the event so received. HandleEvent calls (mostly) event-specific routines, that call (mostly) routines through function pointers attached to MyWindow structures, and take certain generic actions where those do not exist.

GrabEvent works something like this:



It calls WNELo which in turn calls the OS routine WaitNextEvent to actually fetch events. The returned event is then offered to the toolbar (TBEEventFilter), the Editor (PETEEEventFilter) and any special EMSAPI plugin windows (PlugwindowEventFilter).

HandleEvent looks more like this, parceling out events to various event handling functions:



The Editor

Eudora’s editor was written to the Macintosh toolbox some years ago. It handles large volumes of text, and is cognizant of all scripts and languages. The same is not necessarily true for all the glue code in which

it is wrapped; much of that may assume single-byte-per-character.

The editor is compiled separately from the rest of the application, and handles events in its own way.

Editor documentation is, unfortunately, scant at best.

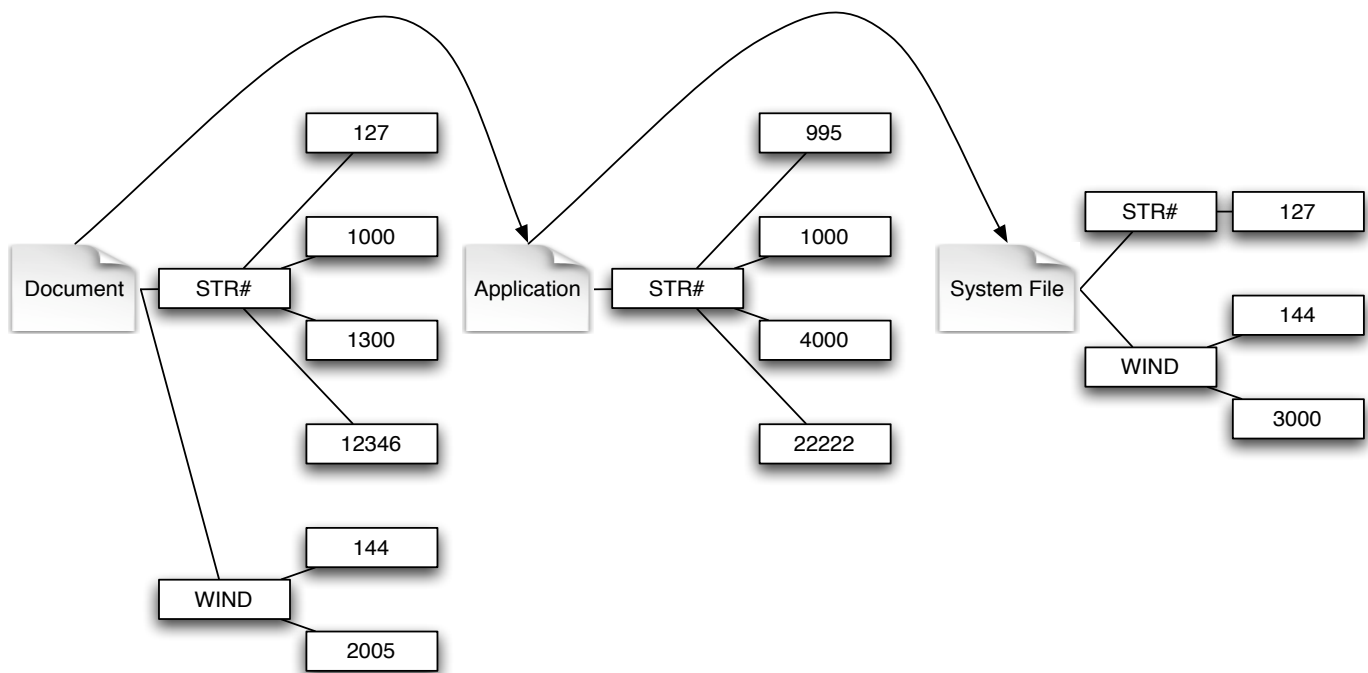
In the long run, limitations on table handling and other advanced layout capabilities probably indicate that the editor should be replaced with something else, probably WebKit.

Strings and Resources

One of the keys to Eudora's usefulness and longevity has been its flexibility. One of the things that gives Eudora flexibility is its use of resources and strings.

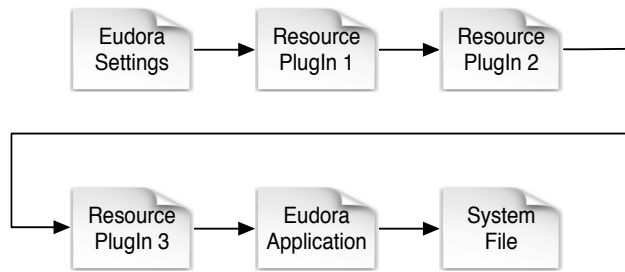
The Resource Chain

One of the interesting features of MacOS Classic is the resource chain. This is an ordered listing of resource files, where the resource in question is searched for in all files from the top of the chain to the bottom. A typical resource chain might be ordered like this:



That is, there is some document open with some resources, and then two resource files that are always open, the application and the system file. Resources “farther up” the chain override those lower down, such that resources with the same type and id that exist in (eg) both the document and system file will be taken from the document rather than the system file. So, STR# 1000 would come from the document, not the application. STR# 4000, on the other hand, appears only in the application, and would come from there. And so on.

Eudora, on the other hand, encourages much longer resource chains, such as (actual resources removed for clarity):



This is useful because it allows simple plugins to change Eudora's resources, and hence its appearance and behavior. These resources can be used to add sounds, settings panels, icons, localizations, and most of all, to change the values of strings. Which brings us to our next subject.

Strings

Strings, specifically strings in resources, are used everywhere possible in Eudora. They are used for user interface elements, of course, to make them localizable. But they are also used for parameter values (such as port numbers), lists of keywords (header field names or html keywords, eg), user configuration items, and even simple dialogs and alerts.

The foundation on which Eudora's strings are built is the STR# resource. This is a resource consisting of a collection of pascal-style (length byte followed by data) strings. These resources can be addressed as a whole (thus referring to an entire set of strings), or the individual strings can be called out. This addressing is done by a simple convention; STR# resources are given id's divisible by 100, and individual strings are addressed by adding numbers to the id's. So a string id of 2043 means the 43rd string of STR# resource id 2000.

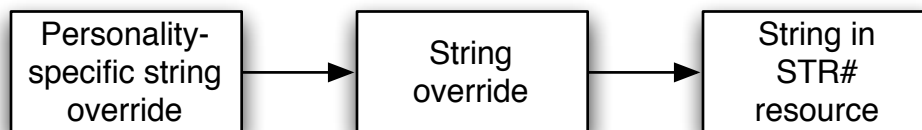
That's the simple story, anyway.

In practice, this is complicated (and made ever so much more useful) by two things; string overrides and personalities.

String overrides are single resources (nominally 'STR' resources) with id's the same as the string id, that are used in lieu of the string inside an STR# resource. So, a string override with an id of 2043 would be used in place of string 43 of STR# 2000, were such a resource to exist.

Personalities are collections of settings that are often used to check or send mail for alternate accounts, or modify the behavior of Eudora for certain classes of mail. These settings are expressed as string overrides, using a resource type that is unique to the personality, consisting of 'ST' and then two numerals, such as '09'.

Thus, strings are actually searched for in this way:



Note that at each stage of the search, the entire resource chain is searched. This allows a vast amount of flexibility. Resources in plugin files can specify specific settings for specific personalities, for example. There is a cache put in front of all this, so that there is very little performance penalty.

Networking

Eudora does not use sockets; Eudora instead uses a simple interface that maps more closely to TCP itself. This interface was originally adaptable to MacTCP, the Communications Toolbox, file i/o, and Open Transport. However, the evolution of the Internet and Mac OS have made both the Communications Toolbox and Open Transport irrelevant. Now, the same adaptation mechanism is used to switch between file i/o, the Carbon Open Transport emulation, and SSL.

This mechanism is simple; it is an array of function pointers called a “TransVector”. This array is changed based on the type of i/o it is desired to do.

Utilities

While the purpose of this paper is not low-level documentation, it’s probably worth talking a little bit about some of the most-used facilities inside Eudora.

GetRString

GetRString and friends are some of the most important routines they are the portal to all the string stuff mentioned above.

```
PStr GetRString(PStr theString,short theIndex);  
long GetRLong(int index);
```

These routines return a string and an integer value, respectively, for the current personality and based on the string id. Learn to love them.

Worth mentioning in this context is the StringDefs file, where you can add a string and an id very simply. Just edit this file, adding a tab-separated set of values, and you will define the appropriate constants and resources automatically.

ComposeString

ComposeString exists in many forms, and is Eudora’s replacement for sprintf.¹

```
UPtr ComposeString(UPtr into,UPtr format,...);2
```

The many other forms include taking format strings from resources, generating alerts, adding to accumulators, etc. Format specifications include string id’s, OSTypes, numbers with units attached, IP addresses, DNS lookups, and more. Provision is made for reordering output items.

ComposeStdAlert

Another handy utility involving strings is ComposeStdAlert, where you may specify in a single string a simple alert. The syntax for the string is:

```
Title text “f” Explanation “f” Other button “f” Cancel button “f” OK button
```

Furthermore, a button name with a trailing bullet (•) is the default button, a trailing hyphen is the cancel button. Any of the texts or buttons names may be omitted. The title and explanation texts are subject to composition with ComposeString.

Accumulators

Accumulators are a way to avoid the inefficiencies associated with dynamically resizing memory. They allow you to grow complex structures in memory without waste or poor performance. Functions exist to add many types of data to them, including composed strings.

Common Idioms and Idiosyncrasies

There are some code usages inside Eudora that may take a bit of getting used to. Forewarned is forearmed.

Loose Pointer Rules

By far the most unfortunately attribute of the current codebase is its use of so-called “loose” pointer rules. The original code used these rules under MPW, where they meant only that pointers to signed and unsigned integral types were treated as equivalent. Unfortunately, at some point Metrowerks decided to change the definition of this compiler flag, and it went from ignoring signedness to ignoring any kind of pointer type-checking at all.

When this change was made, consideration was given to scrubbing the code to remove all the signed/unsigned conversions, but as there were many hundreds of them, this was not done. Instead, we have the unsettling state of affairs where the compiler will NOT do any pointer type-checking.

You should periodically turn OFF the loose pointer rules, and then run the resulting output through the “cwlint” script in MPW. This script will remove the signed/unsigned warnings, and a few other common warnings, and whittle the output down to the point that you may actually find pointer type mismatches.

Cascading “if” Statements

A common coding style inside Eudora is to use multiple “if” statements rather than a list of expressions joined with “&&”. You will often see:

```
if (expr1)
if (expr2)
if (expr3)
if (expr4)
{
    ...
}
```

where a more common style might be:

```
if (expr1 && expr2 &&
    expr3 && expr4)
{
    ...
}
```

Testing the Results of Assignments

Eudora does a lot of this, on purpose:

```
if (x = expr) {...}
```

This is indeed intended to check the value of x after the assignment.

Poor-Man's Default Arguments

It is common practice in Eudora to have two versions of a function; one that takes a full set of arguments and one that takes a smaller set and defaults the lesser-used ones. This is often done using a macro that takes the shorter list of arguments, and a function that takes the full list, with the function named by adding “Lo” to the macro. For example:

```
OSErr CompHeadGetStrLo(MessHandle messH,short index,PStr string,short size);  
#define CompHeadGetStr(p,i,s) CompHeadGetStrLo(p,i,s,sizeof(s))
```

This is often done when new arguments are added to a function, and it's thought best to not disturb the older code.