New Technical Notes

Macintosh

Developer Support

Managerial Abuse

Overview

Written by: Bo3b Johnson August 1988

When using the various pieces of the Macintosh operating system there is a temptation to try to stretch the built-in Managers too far. Developers should be aware of the intended purpose of the various Managers and beware of using them for things that they were not designed to handle. If extended beyond their design goals, they will become slow and unwieldy.

Managers to avoid abusing, and the type of abuse:

1) The Resource Manager is **not** a database.
2) The TextEdit package is **not** a word processor.
3) The List Manager is **not** a spreadsheet.
4) The Dialog Manager is **not** a user interface.

No free database

After using the Resource Manager for a short time, its virtues become apparent: it is very flexible, it is easy to use, it gives disk based I/O with no extra calls, data can be extracted by either name or ID number, and the data is stored transparently so the caller can pretend the data is always available in a virtual memory fashion. With such wide ranging advantages, it would seem that the Resource Manager should be used for everything. It should be apparent that the TANSTAAFL (There Ain't No Such Thing As A Free Lunch) philosophy applies to the Resource Manager as well. If overextended, the Resource Manager will become slow and unusable.

The Resource Manager is not a database, nor is it a good way to store user data. Although it can be used to store very small amounts of data, such as configuration data, and features some of the same characteristics of databases in general, the Resource Manager is a specialized tool designed specifically for the types of things that the Macintosh System needs. Its main virtue for system use is that a large variety of data can be stored on disk, and accessed when needed. This is a primitive form of virtual memory which extends the power of the system beyond what the RAM supplies. Remembering that the Resource Manager was written in an era of 128K RAM, it should be apparent that it is optimized to use as little RAM as possible.

The Resource Manager uses a simple data structure for accessing the data in the file. Examining the Resource Manager file format can show some of the tradeoffs expected. For instance, there is a linearly accessed table which describes all of the possible resource types that are in the current file. Without too much thought it should be apparent that if a file is created with thousands of different resource types then access to those resources will be slow. The reason?

Each access requires scanning a linear array. There is no hashing technique used on the resource types.

There is a similar linear table for the resource IDs themselves. Based on the previous discussion it should also be apparent that if there are thousands of resources of a specific type that the access time will become much larger. It will be imperceptible on a single access of a resource, but for thousands of accesses to the resource file the time spent traversing the linear list will impact the overall speed of the program. The user will not be pleased.

Increasing the slowness by having too many resources as well as too many types will encourage the user to file the program in a ground based circular storage facility.

As stated in M.TB.MaxResInFile, there is a limit of about 2700 resources in a given file due to the way the resources are stored. The performance penalty will arrive sooner, and the dividing line for where it is "too slow" is a personal preference. As a rule of thumb, if the program has the ability to store more than about 500 resources total (both IDs and types), then consideration should be given to using the Data Fork instead. In particular, if the program allows the user to create data files, do not use the Resource Manager to store the user data. The users will always overextend the use of a program. Plan for it, and avoid making obviously bad decisions. For large amounts of data, the File Manager is the place to look. If the program wants to allow simultaneous (multi-user) access with read and write privileges to data files, then do not use the Resource Manager. Because it caches data, the Resource Manager cannot be relied upon as a multi-user database – even for small amounts of data. This is because there is no way to tell the Resource Manager its cache is invalid.

Don't be fooled by a convenient interface. The Resource Manager is not a database, nor is it a file system.

Words to live by

Looking at the TextEdit package can give the impression that there is a full featured word processing system built in. This is even more true now that TextEdit has been extended to support various styles and fonts. Unfortunately, appearances are deceiving, and TextEdit is not up to the job of being a word processor. Looking through the documentation shows that there is a 32,767 character limit on the text in a TextEdit record. The `teLength` is defined as an `Integer`. Another more subtle limit is the drawing limit of the rectangles surrounding the text. The `destRect` and `viewRect` both surround the complete TextEdit record. Using some rather rough approximations, there is an upper limit of about 40 pages of text that can be supported in the QuickDraw rectangle. This is quite a lot for some applications, but is not very many when looking at the job typically required of a word processor. Users do not enjoy breaking their documents into multiple pieces.

There are some other programmatic limitations, not the least of which is performance. TextEdit will become quite sluggish with large blocks of data. After 2,000-4,000 characters

have been stored in a TextEdit record, the performance will have slowed to an unacceptable level. It is notable that the `lineStarts` array is a linear array of offsets into the edit record. If the data towards the end of the data record (high in the record) changes, the offsets have to be changed. This can involve updating thousands of `Integer` offsets for every character typed. If the different font, size and style information is tacked on top of all that, the performance can be expected to suffer with large blocks of text. Make no mistake about it, a full Macintosh style word processor is not an easy thing to write. TextEdit was not designed to handle large

documents. It was designed as a simple field editor for the Dialog Manager, and extended from there. It was never intended to handle the large jobs expected of a word processor.

In order to perform the operations required of a word processor it is necessary to use QuickDraw extensively. The expected Macintosh selection approach with autoscrolling, typing over selected text, cut/copy/paste, and so on are best implemented using QuickDraw directly. How the text is stored internally is the primary determining factor on how the word processor will perform.

Don't be fooled by how easy it is to implement simple editing in an application. TextEdit is not a word processor.

Checking lists twice

The List Manager appears to be a cell oriented display tool, allowing the easy creation of a spreadsheet interface using system calls. The rich interface to the manager makes it easy to handle arbitrary lists of data. Or does it? Although the List Manager is very flexible, easy to use, and general enough to handle graphic elements, its performance becomes unacceptable with relatively modest amounts of data. A one-dimensional list (like the files list in StdFile) can be done very well using the List Manager, but with several thousand items in the list, the performance may not be sufficient. This rarely happens in StdFile of course, and StdFile was the father of the List Manager. Here again, the tool was designed with a specific concept in mind, not to be the ultimate tool for handling any possible arbitrary data. A two-dimensional list of data will become too slow to use with an array as small as 10x100. This can hardly be expected to satisfy the user of a spreadsheet, since one "power" criteria is always the number of cells available.

Why so slow? As above, examining the data structures used by the List Manager can tell a lot about the expected performance and limitations. Notably the `cellArray` used to offset to each cell's data is an old friend, a linear array of `Integer` offsets. It should come as no surprise that inserting or deleting data from the middle of this array is slow. In order to do those functions the List Manager has to update the `Integer` offsets in the array each time. It has to step through each element on the linear array of offsets which will take some time on several thousand elements.

The `maxIndex` field of the ListRec is also notable since it is an `Integer` as well. The lists of data can be no more than 32K bytes in size, which could be somewhat limiting to a user.

In addition, the List Manager is very general purpose, making it necessary for it to protect itself from bad data whenever possible. It needs to check the bounds of any rectangles it uses for example. It tries to minimize drawing out of bounds, so it checks each cell as it is drawn to be sure that it is on screen. Extra validity checks take some small, but finite, time. As the number of elements grows, the time adds up until it becomes a performance problem. Another limitation brought out by the data structure is the `listDefProc`, the list definition

procedure. Since the List Manager is designed to be as general purpose as possible, it was necessary to add the ability to plug in a new defproc. This has ramifications for speed, however, since all drawing has to go through the bottleneck of the defproc. It won't cost much each time, but it will add up over a large number of cells.

In order to get high performance out of this type of display, it is generally necessary to have as much precalculated as possible. This usually means having data structures which maintain

themselves as much as possible, and which do not require changing anything outside of their single cell, thus avoiding impacting the entire display. Linear arrays don't come under this category, since any change impacts all the other cell data in the list. To create a high performance spreadsheet it is usually necessary to go to the QuickDraw level inside of a standard window. It is not typically necessary to be fully general for a specific type of data, so the performance can be improved merely by knowing the type of data expected. To handle large lists of data, the data should be stored in powerful data structures, and displayed with custom routines that know the best way to draw the data.

Don't be fooled by the richness and general purpose interface to the List Manager. The List Manager is not a spreadsheet.

Dialog with the devil

The Dialog Manager is very attractive. It looks like it will handle windows automatically with no programmer intervention, and can handle a wide variety of elements. It seems to handle controls, static text, editable text, and provides a way to display graphic elements as well. It must be the best possible world since the interface is very straightforward, and so much is done for the caller. At last, a superbly general purpose manager that can be used for any interface. Suddenly, reality rears its ugly head again, and it is interesting to note that this free lunch actually requires more work than doing the same job using the Window Manager, QuickDraw, TextEdit, and the Control Manager. Why? There is a hidden cost in terms of getting the Dialog Manager to do exactly the desired task. Here again, if the end result is supposed to be a simple dialog with a few controls, the Dialog Manager is suited to the job. That is what it was written to do. It was not designed as a way to handle the full interface for applications.

As an example of a hidden cost, what if the interface requires that the program be able to handle a disk inserted event? If this is part of a `ModalDialog`, that requires passing a special `filterProc` to the dialog when it is called. It is now necessary to fully understand how the proc gets called, what is legal, and what the proc is required to do. That may not be too hard, but it is time spent on something that has nothing to do with getting the job done; it is only time spent understanding how the Dialog Manager works.

Another example is adding something to a dialog which requires special setup and update routines. Here again, it is not too hard to figure out, but it is time spent trying to tell the Dialog Manager what should be done. There are literally hundreds of these special cases and tough, small problems when trying to extend a dialog past a simple interface. Hundreds of Mac programmers have wasted hundreds (thousands?) of hours finding ways to coerce the Dialog Manager into running a window in a special way.

How about adding a special control to a dialog? Seems straightforward... How about making it modeless instead? How about moving some items in the dialog off screen? How about moving an EditText item off screen? How about wanting to change the dialog template

before the dialog is used? How about all of the above all at the same time?

How about skipping it and using the Window Manager instead?

There are a number of performance penalties for large dialogs as well. A dialog with 50 radio buttons will be unacceptably slow. It should be noted that the Dialog Manager cannot know the desired purpose of the buttons, so it cannot set the button, nor clear another in the same set. In

order to implement the actual radio button aspect of a set of controls, it has to be done by the calling program. At this point, the only thing the Dialog Manager is handling is the creation and drawing of the controls, which can easily be done with `GetNewControl` and `DrawControls`. The Dialog Manager actually gets in the way of a more complex interface. Looking into the data structures shows that the list of items in a dialog is a linear list. Also of note is that there are no offsets to the various items! This is significant because it means that the Dialog Manager has to drive through the entire list of items for every single operation it performs. If it gets an update event it has to traverse the list. If it gets a mouse event it has to traverse the list. This cannot be expected to be fast with 100 items.

Another performance problem for some programmers is the simple drawing scheme used by the Dialog Manager. If a dialog has some items that are offscreen, they get drawn during update events anyway. The Dialog Manager will traverse the list and draw each item, whether it is on screen or not. This comes from the original design of the Dialog Manager, in that it was never intended to handle hundreds of items, or items off screen.

Some rules of thumb: If there are more than 20 items in the dialog it should be a standard window. If a complicated control like a scroll bar is needed, it should be a standard window. If there are items offscreen, it should be a standard window. If there is a pictorial indicator like a progress indicator, it should be a standard window. If it is a modeless dialog it should be a standard window. If any of the items are movable in the dialog, it should be a standard window. If it is necessary to use a `filterProc` to add functionality, it should be a standard window. If in doubt, it should probably be a standard window.

Handling a dialog with the Window Manager is very straightforward, much more so than trying to get around the Dialog Manager. There is the standard main event loop, and a conventional case statement to handle the events of interest. If there are controls in the window, they are easily handled with Control Manager calls. Any special items can be added to the case statement with no tricks. Overall there is more code to write, but the code is much less complex (read as: easier to figure out, easier to debug, easier to maintain). In addition, when extra items have to be added to the window, there is an easy-to-find, logical place to add the code. With the Dialog Manager there may be hidden difficulties.

The Dialog Manager is very powerful, but to use the power it is necessary to use all sorts of hooks, procs, special items, and special calling sequences. As expected, only the interfaces to these things are described in *Inside Macintosh*. The sequence of events is the costly part. For an example of how to add a `userItem` to a dialog, examine M.TB.DialogUserItems. Note that it is not particularly simple to understand. Contrast that with the `FillRect/FrameRect` calls in the code that handles update events in a normal window.

The Window Manager is more powerful than the Dialog Manager. The Dialog Manager uses the Window Manager. The Window Manager is much more straightforward to use since it follows the conventional Macintosh event model. That model is easier to understand and easier to extend. There are more calls to make, but the overall use is much simpler. There are very few special tricks needed to make any conceivable interface in a window.

Don’t be lured in by the “powerful” Dialog Manager calls, tricky hooks, and filter procedures. The Dialog Manager is not a user interface.

Further Reference:

- The Resource Manager
- TextEdit
- The List Manager
- The Dialog Manager
- M.TB.MaxResInFile
- M.TB.DialogUserItems