MacApp 2.0 Memory Management

Any of the following situations can cause an application to stop with a System Error alert, without any chance for that application to gain control:

- Not enough memory to load a code segment.
- Not enough memory to load a PACK resource.
- Not enough memory to save the bits under a menu which is pulled down.
- Not enough memory to load a defproc (WDEF, MDEF, CDEF, LDEF).
- Not enough memory for Standard File to create its file list.
- Add your personal favorite here.

Not all commercial Macintosh applications deal correctly with these issues, in large part because doing so involves some rather complicated code. MacApp contains a memory management mechanism which is designed to help keep a MacApp application from getting into these critical memory situations. It does require some work on your part, but makes it much, much easier to produce a robust application than if you did your own memory management from scratch.

MacApp's scheme works by dividing available heap space into permanent memory and temporary memory (also known as code reserve). Permanent memory is the space occupied by data which your application allocates: objects and any subsidiary data structures you may create. Temporary memory is reserved for your code segments plus any resources and/or memory needed by the Macintosh Toolbox for a short period of time.

MacApp makes sure that your application can't crash for one of the above reasons by always reserving enough space for temporary memory requests to be satisfied. You tell MacApp how much memory needs to be reserved, and it does the rest. It only needs to know whether a given memory request is permanent or temporary. Objects created via New (as well as TObject.ShallowClone and all other MacApp methods which allocate memory) are automatically taken from permanent memory. You can ask for a new handle from permanent memory by calling NewPermHandle instead of NewHandle; for other kinds of requests you set the "permanent" flag by calling PermAllocation(TRUE), make the request, then set the flag back (PermAllocation returns the previous value of the flag as its result). Any other requests (such as those made by the ROM) default to temporary memory.

Note that it is dangerous to have the permanent flag TRUE for any length of time, as any toolbox call which allocates memory or any procedure call which could load a segment will then operate just as it would with no memory management mechanism. For example, calling a procedure in another segment which isn't loaded when the flag is TRUE and there is no permanent memory available will cause a segment loader bomb, even if plenty of temporary memory is available. When debugging is on, MacApp checks for the flag being TRUE in the main event loop or when a segment is loaded and drops into the debugger if it is.

If you need to call a routine which allocates both permanent and temporary memory (such as TextEdit, which allocates permanent data structures and which also loads temporary resources such as fonts), you can do so with the permanent flag set FALSE, and then call CheckReserve afterwards. CheckReserve is a BOOLEAN function which returns TRUE if there is still enough temporary memory. If CheckReserve returns FALSE, you should undo the memory allocation. For convenience, there is also a procedure named FailNoReserve, which calls Failure(memFullErr, 0) if CheckReserve returns FALSE.

Figuring out how much temporary memory to reserve takes some thought. You could specify the sum of the sizes of all your code segments and all the other kinds of memory the Toolbox uses, but that would be wasteful because not all of these items are in memory at the same time. In general, you need to know the sum of the sizes of these items at the point in your application where the largest number of them are in use simultaneously. Some of the Toolbox items mentioned above usually need not be considered in this calculation. For example, the saved bits underneath a menu are only allocated when MacApp calls MenuSelect, which is only done in the main event loop, at which time all the non-resident code segments are unloaded. This is also the case for MDEFs. These will seldom be larger than the code segments which you can load.

There are some situations you do have to watch out for. For example, putting up a standard Open or Save dialog can use up quite a bit of memory: PACK 3 (Standard File), PACK 2 (Disk Initialization), plus the list of files in the current folder which Standard File creates. Also, although the Font Manager will not fail if there is insufficient memory to load a desired font, it will substitute a less suitable font, and your program's screen appearance will degrade. Thus, you should include enough memory to load the largest font you will use in your temporary memory figure. Remember, it will only be used when your Draw method is called, or when you perform text measurement.

Printing is another situation where you can run low on memory. Strictly speaking, it's OK to run out of memory while printing: an alert will come up saying the document could not be printed because there was not enough memory. However, users can find it frustrating if they create a document which is too large to print. If you want to make sure that any document a user can create can be printed, you should factor in the memory taken up by the Print Manager while printing — along with any of your code segments which may be present — when calculating the amount of memory to reserve for temporary allocations.

It is usually easier to split your temporary memory size up into several pieces which you calculate independently. For your code, you may want to use the MacApp 'seg!' resources, which let you list the code segments you want considered (see below). That way, you can just determine which segments are loaded at the time of maximum temporary memory use, and let MacApp figure out their size at execution time. For Toolbox use, you can use some fixed constants. For variable resources like fonts, you can actually alter the temporary reserve size while your application is running.

Depending on how careful you want to be about your application's memory use, you can just pick a comfortably large number (which wastes memory), or you can watch your program in action using the MacApp debugger and MacsBug and figure out the smallest safe number (which is a fair amount of work but gives the best use of available memory).

High Water Mark

The MacApp debugger helps you figure out which set of resources takes the most room by keeping a high water mark for loaded resources. Under the heap & stack command (H), the I subcommand now gives the maximum amount of memory used by loaded code segments, PACKs, defprocs, and so forth, and the R subcommand resets this number to zero. Under the toggle flags command (X), the R flag reports whenever a new high water mark is reached, and if the B (memory management break) flag is set, MacApp will enter the debugger. In calculating this number, MacApp only considers resources on its resource lists (see below).

Note that this won't take into account memory allocated by, say, Standard File or the Print Manager. You should always check the amount of memory used in these situations and any other situation where the Toolbox can allocate large amounts of memory. The best way to do that is to break before and during such a situation, and use the M subcommand of the H command to see how much permanent memory is available (the number labelled "(permanent) FreeMem"). The difference (call it "extra") will be the sum of the sizes of objects you've allocated and those that the toolbox has allocated but which MacApp doesn't track specifically (see "MacApp Resource Lists," below). Also note the set of segments loaded. The sum of "extra" and the "locked resources" number displayed in the debugger, minus any permanent memory you allocated, is the actual amount of temporary memory in use. You can use "extra" to reserve more temporary memory with a mem! resource (see below).

Remember, though, that you are only observing the memory in use at one point in time; memory usage can be greater for a brief period of time, and you won't necessarily catch it in the debugger. Sometimes a trial and error approach is necessary to determine the exact amount of memory being used.

Currently, the most space intensive print driver is the LaserWriter driver. For version 3.1 of the LaserWriter driver, we have empirically determined that "extra" is about 40K. In release 4.0 of the LaserWriter driver, this amount varies, and can be as high as 56K. Moreover, this may well increase in future releases. Fortunately, recent releases of the LaserWriter driver recover gracefully from out of memory conditions, giving the appropriate error message. If you do not need to insure that it is always possible to print, you can eliminate this from your permanent memory reserve.

There is a sporadic bug in LaserWriter driver 3.1 which can cause heap space to be permanently lost. This only occurs when a bitmap font is downloaded to the LaserWriter. Bitmap fonts are only downloaded when font substitution is off in Page Setup (font substitution is always off if you set FractEnable to TRUE or turn off the driver's line layout algorithm) and the user selects Geneva, New York, or Monaco, or if the user selects any other font which is not available in PostScript form (such as Athens or Mobile). The driver finds the largest available size of that font, makes it unpurgeable, then downloads it. Occasionally the driver will not make the font purgeable again, and it remains in memory until the application quits. Since the font is the largest size the driver could find, it takes a significant amount of space (8K for Geneva 24). This bug was fixed in release 3.3 of the LaserWriter driver.

The Low Space Reserve

MacApp keeps a special handle around which it will dispose of in order to satisfy a permanent memory request. This handle is called the low space reserve (it's also sometimes called the permanent memory reserve). You can test if your application is running low on memory by calling the BOOLEAN function MemSpaceIsLow. MacApp makes this test periodically and will call the method TApplication.SpaceIsLow, which you can override to take any action you want. The default version will periodically put up an alert advising the user that memory is low.

Another important issue which the reserve handle helps with is making sure that users don't lose data. Because of the way the Macintosh Memory Manager operates, it makes no guarantee that a particular set of objects which was once allocated in a heap of a given size can again be allocated in a heap of the same size (although it will come pretty close). As a result, you should not allow user documents to grow until they fill the entire heap, since it is possible that you would not be able to read them back in again. Also, if space becomes so scarce that there is no room to create a new command object, users won't be able to decrease their document size, even with a command like Clear.

You can use the low space reserve to handle these problems. If MemSpaceIsLow returns TRUE, don't enable any commands that increase document size, such as Paste, drawing, typing, etc. Always enable commands which allow the user to decrease document size (such as Clear, backspace, etc.). The DrawShapes sample program illustrates this technique. Commands such as Cut and Copy require more thought. On the one hand, they can increase memory use, getting you into the situation where command objects can't be created. On the other hand, if a user wants to decrease a document's size, having Cut and Copy available prevents having to simply throw data away. What you do here depends on your application. For example, UTEView allows Cut when space is low, but not Copy.

If you have a command which allocates additional memory, you should check for space being low at the end of your DoIt method, since otherwise it's possible for the reserve to be in place at the start of the command and completely gone by the end. You should treat running out of low space reserve the same as running out of memory. You can call FailSpaceIsLow, which calls Failure with an error of memFullErr if MemSpaceIsLow returns TRUE. Your command's failure handler should back out any changes it made. Only commands which decrease or don't change memory use should be allowed to eat into the low space reserve.

MacApp itself calls FailSpaceIsLow in several places, including the end of TApplication.OpenNew and TApplication.OpenOld to make sure that a new document doesn't decrease available free space too much. For TApplication.OpenOld, however, MacApp temporarily halves the low space reserve so that existing documents have a little "breathing room" to deal with the nondeterministic behavior of the Memory Manager.

The seg! and mem! Resource Types

MacApp initially sets the size of the temporary memory reserve by looking at all resources of type seg! and type mem!. The sizes of all code segments whose names are listed in any seg! resource are added up as part of the temporary memory reserve. Note that the segment names are the those generated *after* segment mapping. Each mem! resource has three long integer quantities: an amount to add to the temporary memory (code) reserve, an amount to add to the low space reserve, and an amount to add to the size of the stack. MacApp calculates the size

of the temporary memory reserve by adding up the sizes of all segments in all seg! resources and the first number from all mem! resources. It calculates the low space reserve by adding up the second number from all mem! resources, and it calculates the size of the stack by adding up the third number from all mem! resources.

This approach allows a great deal of flexibility. For example, the Debug.r file adds the debugging segments to the list of segments, so that if debugging is turned on there will be room to load them. MacApp defines an initial set of segments, reserves 8K of stack space, reserves 4K of low space reserve, and reserves 4K extra of temporary memory. You can easily add to (or even subtract from, except for the stack) MacApp's values by supplying your own mem! and seg! resources. Remember, however, that these resources only govern the initial value of these numbers. If you wish to change any of them while your program is running (except the stack size, which can't be changed), you will have to call the routine SetMemReserve (see the MacApp source code for details).

Remember to factor in temporary memory taken up by things other than resources (such as the memory taken by the Print Manager while printing). For example, the MacApp debugger may tell you that the largest set of resources loaded at one time occurs when opening a document and totals, say, 130K. When printing, your resources may only total 110K. However, the LaserWriter driver uses another 40K of temporary memory, so your maximum temporary memory use is 150K, while printing. In that case you would list the segments you use during printing in your seg! resource, and use a mem! resource to reserve an additional 40K for the Print Manager.

Commit Methods Must Not Fail

Note that if your command object overrides Commit, it is vital that it not cause a Failure. MacApp must call the Commit method of the most recent command (if it has not been undone) in order to save the document or quit the application; if Commit fails, your user will be really stuck, unable even to exit the application. The MacApp command architecture assumes that by the time Commit is called, the command was successful. There are three ways of dealing with this problem.

The first and best way is to set up your data structures in such a way that your Commit method doesn't need to allocate any memory (or increase net memory use). Of course, this isn't always possible. The second way is to preallocate the memory which your command needs to Commit in your DoIt method. Then, when Commit is called, free this memory and proceed. The third way is to allocate the memory you need to Commit from the temporary memory pool instead of the permanent memory pool. You should only do this if your Commit method's memory use has an upper bound, and if the memory will be disposed of by the end of the Commit method. If you do this, you must make sure that your temporary memory reserve is big enough to accommodate this memory use (you can use a mem! resource to do this).

For an example of how subtle these considerations can be, look at the Paste command in the DrawShapes sample application, whose Commit method can actually fail under certain circumstances. The Commit method moves the shapes which have been pasted from the list of "virtual" shapes associated with the command onto the actual list of shapes for the document. It does this by repeatedly deleting a shape from the virtual list, then inserting it into the actual list. Although this seems safe because it does not cause net memory usage to increase, it can still fail, because it may not be possible to grow the actual list's handle by four bytes even though another handle has just shrunk by four bytes and the heap may be completely unfragmented. This is just a consequence of the way the Macintosh memory manager works.

Two possible solutions to this problem: grow the actual list in the DoIt method so that the memory is already allocated, or perform the transfer of shapes with the permanent allocation flag off (currently this is not possible without overriding TList). Note that this latter option is safe because net memory utilization is not increasing; by allowing the Commit method to briefly eat into temporary space, we are just giving the memory manager a little more "breathing room" in which to rearrange the heap.

Although your user won't get stuck if your UndoIt or RedoIt method fails, he or she will probably get upset. The feeling of safety and confidence which users get from having undo available will vanish the first time they try to use it and it fails. The message they get is "Undo doesn't work." Be nice to your users, and make sure that your UndoIt and RedoIt methods can't fail either. A less desirable alternative is to detect in your IMyCommand or DoIt method that Undo won't work and put up an alert that the command will not be undoable. Do this before your

command makes any changes, and give the user a chance to cancel the command. If the user says OK, remember to set fCanUndo to FALSE in your command so Undo will be disabled. If the user says Cancel, you can cause your command to fail without putting up an error message by calling Failure(0, 0). MacApp uses this technique itself to handle Cancel choices in dialogs.

One more hint for commands: try not to allocate any memory in your IYourCommand method. There is nothing actually wrong with this, but since the previous command has not yet been committed and freed, and the Undo Clipboard might still be around, it's more likely to fail. If you allocate your memory in your DoIt method, more space will be available.

MacApp Resource Lists

MacApp keeps a list (actually several) of the resources which it considers "temporary:" that is, they are considered as not taking up permanent memory. It constructs this list at application startup time, and uses it whenever it calculates how much temporary memory is in use and how much more to reserve. These resources are the ones that the MacApp debugger describes in the H command. MacApp will also purge these resources when trying to satisfy a temporary memory request unless they are locked.

By default, this list contains all resources of type CODE, PACK, LDEF, CDEF, WDEF, and MDEF (except those which come from ROM, or reside in the System heap). Normally, you don't need to worry about this list at all. It's OK for temporary resources to not be on the list, although they will be purged frequently when space is low, and they won't be figured into the resource statistics in the debugger. If you have such resources (fonts, for example), and if you want to reserve some memory for them, you may want to consider adding them to the list. Look at the UMemory unit to see how the lists are managed.

If you do put fonts on the list, you should move them high (with MoveHHi) and lock them when you do so to prevent MacApp from purging them; the font manager expects that fonts marked nonpurgeable won't be purged.

Segmenting Your Application

In order for the memory management mechanism to work properly, your application must be segmented properly. The more code which is unnecessarily dragged into memory, the larger your temporary memory reserve must be, and the less space is available for your user's data in any given memory configuration. MacApp defines the following code segments for your application:

ARes	Resident application code. Anything that gets called frequently in the main event loop (such as your DoSetupMenus methods), drawing, or typing.
ADebug	Your debugging code, i.e. any procedures or methods only present when debugging is on.
AFields	All of your Fields methods should go here.
AInit	Code which you only use at application start-up time (IYourApplication).
ATerminate	Code which you only use at application shut-down time.
ASelCommand	Code used to select the next command (DoMenuCommand, DoMouseCommand, DoKeyCommand, and IYourCommand methods).
ADoCommand	Code used for executing commands (all other TYourCommand methods except IYourCommand).
AClipboard	Clipboard code that is not part of a command (MakeViewForAlienClipboard, GivePasteData, WriteToDeskScrap, but NOT TYourPasteCommand).
AOpen	Code used when opening (DoMakeViews, DoMakeWindows, IDocument, DoMakeDocument, IDocument, IView).
AClose	Code used when closing (TYourDocument.Free, FreeData).
AReadFile	Code used when reading or reverting (DoRead, ShowReverted, DoInitialState).
AWriteFile	Code used when writing (DoWrite, DoNeedDiskSpace, SavedOn).
AFile	Code used when reading or writing (typically you won't have anything in this segment).
ANonRes	Catch-all non-resident segment, for infrequently used methods (e.g. ResizeWindow).

The default segment mapping established by MacApp combines these segments (except for ARes) with the corresponding ones for MacApp. You can override some of MacApp's mappings by specifying your own mappings in your make file, and you can override MacApp's mappings entirely by placing a definition for the Make variable SegmentMappings in your make file (Make will issue a warning, but your definition will override MacApp's). Look at MacApp.make1 for guidance.

Even using the suggested segment mappings, you may overflow the 32K bytes limit on the size of a segment. You can override MacApp's segment mappings in your application's .make file by providing your own -sn mappings; the Linker gives your segment mappings priority over the ones MacApp specifies.

You may want to make some application code segments resident (i.e. never unloaded). This is the case with ARes, for example. To do this you can use the 'res!' resource (see below) or call SetResidentSegment (generally after calling InitToolbox, InitPrinting, etc. but before doing New(gYourApplication)) in order to make ARes resident. Many of the sample programs, including DrawShapes and DemoText, use the 'res!' technique.

Generally, there is a trade-off involved in segmentation. A few large segments make your program run faster, but increase its memory requirements. More, smaller segments decrease memory requirements (and are a necessity to run in a small Switcher partition), but make your program slower to start up. This latter problem can be alleviated by using the JumpStart utility included in the Macintosh Development Utilities product, available from APDA.

If you're really desperate to decrease segment sizes, you can change MacApp's segmentation by editing the source code and increasing the number of segments. This is not recommended. Be extremely careful about which segments are resident if you do this.

The res! Resource Type

The 'res!' resource type is used to identify code segments that are to be made resident (i.e. never unloaded). It's format is similar to the 'seg!' resources in that each 'res!' resource consists of a list of segment names. When MacApp initializes the application it makes resident any segment listed in any 'res!' resource found. 'res!' resources are already included for MacApp's resident segments. You can add a 'res!' resource to your application's resource file to list resident segments you've defined. Note that as with 'seg!' resources the segment names in the 'res!' resources are the names *after* segment mapping.