NOTE: This Technical Note has been underlined retired. Please see the Technical Notes page for current documentation.

Technical Note PT21
MacApp Segmentation Illuminations

This Technical Note describes MacApp segmentation strategies and guidelines. It also describes performance, runtime, and development tools issues related to segmentation. Some of the discussion is also relevant to general segmentation strategies with non-MacApp-based applications. The MacApp techniques are based on MacApp 3.0; however, many of the issues are also relevant to MacApp 2.0.

[Dec 01 1992]

Introduction

Some people consider defining a segmentation strategy for a Macintosh application a black art. Well, it is a form of programming art. Like many problems in software development, defining a segmentation strategy for a Macintosh application requires choosing between a number of conflicting tradeoffs to meet the performance criteria of a given application. Segmentation strategies that are optimized for speed generally require a larger footprint in memory, while strategies that are optimized for memory usage come with a performance hit. Unfortunately, there are no tools available for automatically segmenting applications.

There are few tools available that help define segmentation strategies. Also the information on how to do this is not fully documented.

This Technical Note will try to cover the most important issues of segmentation with MacApp programs and to help developers create their own segmentation strategies.

Back to top

The Need for Segmentation

Inside Macintosh describes how the Segment Loader works, and why there's a need to create segments that are loaded into memory on demand. The syntax for creating a segment name is:

C++: Specify a `#pragma segment SegmentName` in front of the member function

Object Pascal: Specify a `{$S SegmentName}` in front of the method

If you forget to place the segment compiler directives in your method, it will inherit the earlier directive (in C++ as well as in Object Pascal) all the way to the end of the file, so suddenly you'll find many methods inside one of your segments. Methods without any defined segment will go into the `Main` segment, which could get really crowded after a while. So, check your segmentation directive for each method. For instance MacBrowse has a function for doing this that shows what each segment contains. Also, the `Link -map` (or the `MABuild -LinkMap`) flag creates a link map with information

about what functions belong to what segments.

Segments are really `CODE` resources in disguise, so MacApp is able to control the purge and lock bits on segments just as it does on handles or objects (as in `TObject`'s `Lock` and `UnLock` methods). A locked handle is also unpurgeable, so you don't need to worry about purging once you have locked the object in memory. MacApp marks all code segments as nonpurgeable so that the MacApp memory manager can control when and which segments it purges in low memory conditions.

Methods are the actual routines stored in the `CODE` resources; data is stored either on the stack, in the application heap, or in the specific part of the heap that is the A5-world. In many cases, calling a method whose segment is not currently stored in memory causes a segment load to occur, which might cause heap blocks to be moved in order to locate a place to put the new segment.

> **Note:**
> This is one reason calling a new class method can suddenly trigger memory dereferencing bugs.

Back to top

Jump Tables and Performance

There is a known relation between jump table sizes and segmentation. For normal procedures and functions, we don't need a jump table entry if all calls to the routine are from the same segment (*intrasegment call*). However, we need jump table entries if there are calls to other segments from the routine (*intersegment calls*). Examine the segmentation of your code; you might find places where a change in segmentation would eliminate jump table entries. With some effort you may shrink big jump tables and improve the performance of your application.

> **Note:**
> Object-oriented code relying on polymorphism makes this approach nearly impossible. This is because we will never know what function is finally called via the virtual or method table dispatch.

Some programmers worry that many C++ accessor (Get...) and mutator (Set...) methods will increase the jump table entries considerably, but you can avoid this by using `C++ inline` functions. Anyway, if your classes have too many field change and access methods, perhaps it is time to examine the object. Is it really a structure in disguise?

Caching of member field values inside the class (for instance keeping track of field values using temp variables) decreases the need for Get and Set calls. This design is, however, not purely object-oriented, because the class then needs to know about the internal implementation of an object. For instance it needs to know when the cached value is invalid, and it also assumes too much about the internal fields. In addition, we could place the major parts of an object inside one single segment for further performance improvements. This helps mostly if the accessors are providing information to other objects that reside in the same segment, as in the case where we use accessors internally in the same object. You can use `dumpobj` to dump the object file and find information about each segment.

The Segment Loader has to fill the jump table with the right addresses when the segments are loaded in. When the segment is unloaded, the environment has to reset the jump table with information about the missing segment. MacApp has to make sure that memory is always available for data and unloaded segments. All this takes time, so clever segmentation does improve performance. Also, PowerBook owners don't like applications that spend a lot if time starting the hard disk--for instance for fetching CODE resources frequently! For example, if we place functions that call each other in the same segment, we will eliminate other segment loading events.

Back to top

Strategies

One strategy for organizing segments is based on functionality--functions that work together *should be placed in the same segment* (see reason defined earlier). For example, we need certain routines during an application's initialization phase, but after initialization is complete they can reside on disk until the next launch of the application.

Another strategy is to organize segments so they are *as small as possible*. This means that the application heap will contain only those segments that we need, increasing the amount of application heap available. The problem with this segmentation strategy has to do with all the Segment Loader calls that we trigger every time a function is not available in memory. This happens only if the segment itself is not initially loaded. Once a segment is loaded, it is marked purgeable when not used by MacApp, and unless we have a full application heap, the segment is still present in memory. Still, if we need to load a lot of segments from a hard disk, it will cause a lot of disk spinning in the case of portables. And end users don't like disks that spin, because they decrease the battery lifetime.

Back to top

How to Segment MacApp Code

The strategy presented below is a guideline to MacApp application segmentation, but it should not be taken as a prime directive. However, as this is the current MacApp strategy it is not really feasible to use any other alternate segmentation strategies with MacApp applications, because MacApp will be a big part of the application code. However, feel free to experiment and test with the other alternative--small CODE segments--in code that you control.

The MacApp strategy is based on functional groupings, where functions related to each other (they call each other) are placed in the same segment. The functional grouping is:

SEGMENT NAME	USE
Main	Main resident segment, will contain common code (libraries usually).
RRes	Routines you call often, or resident routines. DoSetupMenus DoSetCursor DoIdle Draw DoKeyCommands (if typing)
RNonRes	For routines you rarely call, or nonresident routines.
RDebug	Debugging code.
RInit	Functions used only once in the program. IYourApplication Initialize DoPostCreate
RTerminate	Functions used only when you quit the application.
RSelCommand	For selecting commands (TCommand). DoMenuCommand DoMouseCommand DoKeyCommand (if not typing) TYourPasteCommand IYourCommand
RDoCommand	For performing commands, like TCommand objects. TRecolorCmd::DoIt TSketcher::TrackMouse TTypingCmd:Commit
RClipBoard	For Clipboard noncommands. MakeViewForAlienClipboard GivePasteData WriteToDeskScrap
ROpen	Opening documents and data structures. DoMakeViews DoMakeWindows IYourDocument DoMakeDocument IYourView
RClose	Closing documents and data structures.
RReadFile	Reading from disk. DoRead DoInitialState
RWriteFile	Saving files to disk. DoNeedDiskSpace DoWrite
RFile	TFile based functions, rarely used.
ConstructorRes	Empty constructors (those that are not used by MacApp. Stack based classes that have working constructors should be placed in the same segment as the I<method> or the Initialize method.
IteratorRes	MacApp iterators.
MADebugger	MacApp debugging code (code generated by qDebug, qUserFlag[1-3]).

MacApp has far more segments. We recommend that you do a search of the "`pragma segment`" string in the MacApp library sources to get the whole picture of what function groupings MacApp uses. Or use MacBrowse or a similar browser that shows segmentation information for each method in the framework. For instance, if you write `TAppleEvent`-based member functions, check the `UAppleEvents.cp` file to see where various `TAppleEvent` member functions are placed. If you override any of them, make sure that you place the overridden method in the same segment as the original one. For instance some of the Apple event-related functions and methods should be placed in resident segments; otherwise you could get into trouble. Here MacBrowse also gives good support for a quick lookup of segment names.

Other Issues:

Unsure about the placement?

Place the method in the same segment as other methods that call it.

Stack-based C++ classes?

Place the member functions of these classes in the same segment that corresponds to the functionality (in the same segment as the routine or member function that use them).

WDEFs, CDEFs, and similar code segments--should they be part of the class segment?

Place these CODE segments into the `Main` segment. This is because we want these to be always resident in memory for fast access.

> **Note:**
> Monitor the size of the `Main` segment.

Back to top

'res!' and 'seg!' Resources Explained

MacApp programmers sometimes will get puzzled concerning the use of `'res!'` and `'seg!'` resources. The `'seg!'` resource defines those segments that are loaded into memory when the program is making maximum use of memory. MacApp uses this information when keeping track of the code reserve to ensure there is room for the `'seg!'` code

segments at the maximum point of memory use. See Chapter 6 of the *Guide to MacApp Tools* concerning discussion of maximum use of memory.

The `'res!'` resource defines those segments that are always resident in the heap. (Segments are made permanently resident via a global function called `SetResidentSegment`.)

One use for making segments permanently resident is for time-critical functions grouped together in a special segment. Thus, loading the segment doesn't require overhead if the method is suddenly needed. For example, we could use this technique to reduce overhead for time-critical communication methods. Also, we might have functions that are called during interrupt time that always need to be resident.

Here's an example of a `'res!'` resource defined in the resource file:

```
resource 'res!' (kMyMacApp,     purgeable) {
    {    "AWriteConn";
         "AReadConn";
         "APoll";
#if qInspector && !qDebug
         "GDebugConn";
#endif
#if qPerform
         "GPerformanceComms";
#endif
    };
```

MacApp will also place internal trap patching code in resident segments, to avoid runtime problems with patches that some entity might suddenly unload. This is also true with programmer-provided trap patches.

Back to top

UnloadAllSegments Explained

The MacApp programmer should be aware that segments live a dangerous life inside the MacApp framework. Unless they are resident, or needed, they are purged out. This happens both at initialization time, as well as during the lifetime of the application. We don't want to have segments floating around in memory that we need only during the initialization phase of the application. Likewise, we don't want to have segments in memory that we trigger very few times during the application lifetime.

So who is taking care of the segmentation cleanup? Your mother? No, it's a global function called `UnloadAllSegments` that is called both during the initialization phase and during the idle time of the application. It is also triggered when the application starts to have little memory space. `UnloadAllSegments` purges code segments from the memory unless they are locked, in use, or resident (specified in the `res!` resource).

However, we might have situations where we have written code by mistake so that `UnloadAllSegments` wants suddenly to purge the segment that is currently in use. This function could unload a segment containing a method that you will return to later. Your stack currently references a method contained in the segment that `UnloadAllSegments` wants to unload. Danger, danger.

Now, if you have SourceBug running in the background, and have the warning flag in the Debug menu entry set for unloading segments, you will get a nice little warning from `UnloadAllSegments` before you get into the binary Armageddon...

Back to top

Development Tools, ModelFar Support

For a long time, segments were restricted to 32K sizes due to the A5-relative data referencing with 16-bit offsets, but MPW 3.2 eliminates this 32K limit on segment size via new switches to the compilers and the linker.

The 68020 introduced 32-bit PC-relative branching (`BSR.L` statements), but that didn't help the Classic and other 68000-based Macintosh computers. Instead, MPW 3.2 makes use of branch islands. This simple, elegant idea is based on the implementation of PC-relative code-to-code references. The linker splits a large code segment up into smaller 32K areas by inserting branch islands. These branch islands serve as intermediate points that are within range of PC-relative jumps, thus making it possible to make a call across a segment that would otherwise result in a larger-than-32K jump.

Another new feature is "32-bit everything," which transparently removes the limitations on code segment and jump table sizes and the size of the global data areas. The drawback is a larger code size footprint and some slowdown due to increased load time for the larger code segments. But hey, look what you get!

We activate 32-bit everything code generation and linking by using the `MABuild -ModelFar` flag (supported from with MacApp 3.0 forward).The new MPW 3.2 documentation on the runtime architecture explains the implementation. The trick is that the compilers generate instructions with 32-bit addresses (instead of the normal 16-bit offsets), and that these 32-bit addresses are relocated at load time by the segment load address or by the contents of A5, as appropriate.

Finally, one can generate larger than 32K jump tables using the `-wrap` option. This uses unused space in the global data area for additional jump table entries when it starts to get crowded inside the 32K segment. However, at best this utility doubles the jump table size, and if your global data area is already filled with data, you're out of luck.

Note:
You can't use both the `-wrap` and the `ModelFar` flags in order to cheat and create a large global data segment with jump table entries as well. Use either flag only.

See MPW documentation for more information about `ModelFar` and runtime issues.

Also, the MacApp debugger has support for monitoring segment loading and unloading. Please consult the current MacApp documentation for more information.

Back to top

Segments and Fragmentation

Sometimes we want to move a separate segment back into the `Main` segment, to avoid too many segments--a condition that can lead to heap fragmentation. This could be the case with libraries that we use with the MacApp application. For instance functions from libraries might reside in small segments, and we would like to remap them back to a resident segment, such as `Main`. The MPW `Link` and `Lib` tools can remap the segmentation with routines to other segment names. The syntax looks like this:

```
Link [options...] objectfile... >= progress.file
    -sn=oldSegName1=newSegName1
```

Note that fragmentation is not a problem for MacApp applications. All resident segments are preloaded and locked high in memory. Nonresident segments that are in use are also locked high. Segments that are not in use but still present are not locked. This means they can be moved by the Memory Manager (which does not contribute any fragmentation).

Another solution is to use the linker to mark code resources from the libraries as locked. These segments will then be loaded into memory as resident segments, avoiding fragmentation problems. To do this, for instance modify the user variable `OtherLinkOptions` in the `MAMake` file:

```
OtherLinkOptions = [[partialdiff]]
    -ra FOOSEGMENT=resLocked [[partialdiff]]
```

Yet again in the case of MacApp this is not necessary. The MacApp way is to define these segments in the `'res!'` resource, and MacApp will lock the resources at application startup.

Finally, you can use the linker to merge old segments into new segments with the `-sg` option:

Back to top

Linker Problems and Errors

A very typical `Link` error with huge MacApp projects looks like:

```
### Link: Error: 16-bit reference offset out of range.[....]
### While reading file "Work:Sources:FooBar SuperApp:[....]
### Link: Error: 16-bit reference offset out of range. [....]
### Link: Error: 16-bit reference offset out of range. [....]
```

This means that the linker can't create 16-bit offset references within the same code segment, or that we can't do 16-bit PC-relative jumps. The way to go is to start using `ModelFar` support.

Sometimes even if we compile and link using `-ModelFar`, we will still see these problems. This has to do with `Runtime.o` and all other standard libraries in MPW that are not compiled with `ModelFar`. Sometimes the placement of link modules helps, where the module with the main stub is the first one, `Runtime.o` is the second one (usually `Runtime.o` and `Main` talk with each other a lot), then most standard libraries and all the other modules last. If you suspect this is the case, modify the `"Build Rules and Definitions"` MacApp file, and move the `Runtime.o` link instruction nearer the libraries or object files that contain `Main` segments.

Another trick is to use the `-sortsg` link flag. The linker will move the most frequently called routines to the first 32K part of the segment, and in some cases this will solve the offset problem. Finally if you have a lot of string literals, if you use the `-b2` link flag the strings will be embedded in e.

Back to top

Resident Segment Dangers

Here's an important note to remember. If you make a segment resident (for instance including it in the `'res!'` resource), then the MacApp memory management code will call `GetNamedResource` to load the segment, and also lock it. However, the jump table still has this segment defined in an unloaded state.

So the first time you call a routine in the segment, the code will trigger a `LoadSeg` call. Now the jump table has the segment defined to be in a loaded state (no more need for `LoadSeg` calls).

This works fine during a safe period of segment loading. However, if you call the routine during a critical time when it's not safe to call `LoadSeg` (for instance from a VBL task), you might get into trouble.

One workaround is to make sure to call the first routine from this resident segment during a safe time. For instance you could call the function itself from another segment (as in the initialization phase, from `Main`). This will cause the Segment Loader to convert the function's jump table entry, so this function could be used at more dangerous times. You need to define a specific way to inform your function that it's only been called to load its entry, and not to do any useful work (until later).

Back to top

Conclusion

It is hoped that these suggestions will improve the segmentation strategies in your application. Future new runtime models (the Shared Library Manager, PowerPC runtime environments, Bedrock) might radically change the way developers segment code. Stay tuned for more news.

Back to top

References

Inside Macintosh, Volume II, Chapter 2 (Segment Loader)

MPW Command Reference (forthcoming)

Building and Managing Programs in MPW (forthcoming)

MPW 3.2 Release Notes

Guide to MacApp Tools (MacApp 3.0)

MADA 92 Orlando Conference CD, Presentation on the MacApp 3.0

Memory Manager

Back to top

Downloadables

Acrobat version of this Note (K).   Download