New Technical Notes

Macintosh

Developer Support

MacApp Q&As

Platform & Tools **M.PT.MacApp.Q&As**

Revised by: Developer Support Center October 1992
Written by: Developer Support Center October 1990

This Technical Note contains a collection of Q&As relating to a specific topic—questions you've sent the Developer Support Center (DSC) along with answers from the DSC engineers. While DSC engineers have checked the Q&A content for accuracy, the Q&A Technical Notes don't have the editing and organization of other Technical Notes. The Q&A function is to get new technical information and updates to you quickly, saving the polish for when the information migrates into reference manuals.

Q&As are now included with Technical Notes to make access to technical updates easier for you. If you have comments or suggestions about Q&A content or distribution, please let us know by sending an AppleLink to DEVFEEDBACK. Apple Partners may send technical questions about Q&A content to DEVSUPPORT for resolution.

New Q&As and Q&As revised this month are marked with a bar in the side margin.

MacApp developer technical support options
Written: 10/27/90
Last reviewed: 2/20/91

Is there a special address for MacApp technical support or is MacDTS the correct target?

—

MacDTS an option, though there are also other avenues of tech support for MacApp programmers as well. The first is the group address MacApp.Tech$. This is a group of MacApp developers on AppleLink that ask and answer questions of each other. You can simply send your questions to that address and request replies back to your personal account, or you can join the group and receive copies of all the mail sent to it by sending a request to MacApp.Admin.

Another avenue of support is from the MacApp Developer's Association. This is a non-profit organization set up by MacApp lovers. It's got about 1500 members across the nation, and even has a European counterpart. It has an annual MADA conference and a bimonthly magazine. If you are interested, call them at (206) 252-6946, or write to them at AppleLink

address MADA. Annual fee is $75.

Debugging options for C++

Written: 11/28/90
Last reviewed: 12/19/90

What is the C++ equivalent for the {$D±} Object Pascal compiler directive? I'd like the same debugging options for C++ that exist for Object Pascal.

—

The MacApp debugger is supported with C++ Object Pascal object information. In other words, you can't trace other objects than those derived from PascalObjects.

This works OK if you include a "-trace on" or "-trace always" with CFront when you compile your MacApp source code. We assume that you are using the latest MPW C++ (3.1 final and MPW 3.1ß C tools). Use either the startup file and include this as one of you basic parameters, or send it from MABuild.

We did some testing, and it seemed to work OK. In general the S (stack info) command with the MacApp debugger has a lower signal/noise ratio than the R (recent PC) command concerning useful information, but both work OK.

Extracting text from MacApp TEditText views in dialogs
Written: 6/1/91
Last reviewed: 8/1/91

Where/when is the best place to extract text from my fill-in-the-blank Macintosh dialog with several TEditText views? Does a method get called when another field is selected or another view (like a button) is chosen? I can think of several ways to accomplish what I want to do, but I'd rather use the MacApp view architecture the way it is DESIGNED to be used.

—

You can get one answer to this by looking in the DemoDialogs example (C++ or OP). TView has a method called DoChoice, which should be activated, for instance, when a button, check box, or even with TrackMouse operating over a particular view or groups of view, has triggered something, and the application wants something to be done. Using MacBrowse, do a "Find References" (or Command-R) to find all references wherefrom DoChoice is called, such as TrackMouse or DoKeyCommand. Fields are Views, so they will inherit DoChoice, and in the case of TEditText you could, for instance, call DoChoice from HandleMouseDown (when clicking in a particular field).

HandleMouseDown is also a TView method, so most views have this one implemented. Special sub-Views have additional indicators that something was triggered inside the view.

%_BP and %_EP routines
Written: 6/17/91

Last reviewed: 8/16/91

When I attempt to link my program, the linker complains about %_BP and %_EP being undefined. What are these routines? Why are they needed? Why are they not being found?

_____

The functions %_BP and %_EP are special routines that are conditionally called at the beginning and end of each procedure or function. This is designed for use by the MacApp debugger, which uses these calls to follow the execution of the code being debugged. You can also put them to any purpose you wish, such as profiling.

Whether the calls to these routines are inserted is controlled through compiler directives: in MPW C, you can use the #pragma trace on|off directive to specify whether you wish these calls to be inserted. This can also be done with the compiler option -trace on|off|always|never, which makes the default state either on or off (the always and never options also override the #prgama trace directives). In MPW Pascal, you can control this feature with the {$D++} (which turns the calls on) and {$D--} (which turns them off) directives.

Generally, the routines themselves are supplied by a library you link with, such as the MacApp debugger or a profiling tool, so you won't need to write your own. If however, you decide to use this feature for your own devious purposes, here's how.

The procedures called are passed no arguments, and so should be defined as follows:

in MPW C,

```
pascal void BP()
```

or in MPW Pascal,

```
PROCDEDURE BP;
```

The only difficulty is that neither C nor Pascal allows you to create a function that begins with a % sign. Thus, you need to explicitly tell the linker to treat your functions, BP and EP, as %_BP and %_EP. To do this, add the options -ma BP=%_BP -ma EP=%_EP to your link statement, and all will be well.

Saving window state in pre-3.0 MacApp

Written: 8/23/91
Last reviewed: 9/17/91

How do I save new window size and location in a TWindows resource so it can be reused next time this window is opened or created?

—

To save your window state in an MacApp application, your implementation needs to do the following:

1. Create a record which holds the window state
2. Track whether the document is being opened or reopened
3. Read the window state information in your DoRead method
4. Restore or initialize the window state in your DoMakeViews method
5. Add in the number of bytes required for the window state when you compute the disk space required for the document file.
6. Write the window state when you save the document

You could store the window state record in the resource fork of the document file. Here’s a simple example:

```
TYPE
   DocState   = RECORD
        theLocation       : VPoint;
        theSize           : VPoint;
   END;

   HDocState = ^PDocState;  {Handle to DocState information}
   PDocState = ^DocState;   {Pointer to DocState information}
```

In the document class, add some fields for keeping track of the state, as in the following code:

```
   fDocState : DocState;  {the record with the window state information}
   fReopening: Boolean;
```

The fReopening flag is used to test whether the window state needs to be restored. If this is true, DoMakeViews should then know that it needs to restore the window state from the fDocState record. You need to set this boolean false in the Initialization phase of the Document object.

In the DoRead, check if the resource fork is open, and read in the record to the document field, and set the boolean to true.

Inside DoMakeViews, call another special method called RestoreWindow (or something similar) which takes the information in fDocState, and acts upon thewindow. For example, for window location, call the window Locate method and provide Locate with the values from the fDocState record. Then force the window on the screen. Same with the scroll location, call the ScrollTo, and provide any possible VCoordinates from the fDocState. Same also with the window size, use TWindow.Resize and resize the window according to the saved values.

Finally, you need to save the TView.fLocation, TView.fSize and other values in the fDocRec record, calculate how much space you need in DoNeedDiskSpace:

```
   rsrcForkbytes = rsrcForkBytes + kRsrcTypeOverhead
                                 + kRsrcOverhead + sizeof(DocState);
```

And in DoWrite you add the resource to the resource fork, and before that catch the latest values in the various fields:

```
   ...
   docStateHandle := HDocState(NewPermHandle(sizeof(DocState)) );
   with docStateHandle^^ DO
     BEGIN
       theLocation := aWindow.flocation;
       ...
     END;

   AddResource(Handle(docStateHandle), kDocRsrcKind,
                                       kDocStateID, 'Doc State');
   FailOSErr(ResError);
   ...
   END;
```

All this is slightly different in MacApp 3.0, because locations are in VPoints, and windows are keeping track of the location and size directly (fields in TWindow), so you need to modify the source code for this. MacApp 3.0 DrawShapes will also show how to save the window state information.

The LACS example in develop also has code that shows how to store and restore the window location information.

X-Ref:
“Asynchronous Background Networking on the Macintosh,” develop #2:1, Winter 91

MacApp libraries: Building debug versions

Written: 11/18/91
Last reviewed: 12/12/91

I’m having problems building Debug versions of the MacApp library (“-NeedsFPU -Debug” and others). Particularly, with the UDebug.p unit, and its references to “Notification” structures. Is there a fix?

—

Assuming that you are using MPW 3.2, the solution to your problem is documented in the MacApp 2.0.1 Release Notes dated September 28, 1990, located on the E.T.O. #5 CD. On page 18 is the following excerpt:

There are several incompatibilities between MPW 3.2 and 3.1. If you wish to use MPW 3.2 with MacApp, in the file “Startup”, comment-out the line:

```
SET MABuildDefaults ""{MABuildDefaults} -d qMPW31=TRUE"".
```

This should work with beta and final versions of MPW 3.2, although it’s only been verified with MPW 3.2b3, which was available at the time.

However, if you are developing on the Macintosh Quadra computers with MacApp 2.01 or MacApp 3.0 (versions prior to b3) and are using -NeedsFPU or -Debug you may have a problem with crashing. The workaround is to comment out the following lines in procedure InstallInterceptors found in UDebug.incl.p, which installs LineF exception handlers:

```
pOldexLineF := ProcPtrPtr(exLineF)^;
ProcPtrPtr(exLineF)^ := @XDebugLineF;
```

Versions beginning with 3.0b3 no longer have this code.

The reason for commenting this code out is that the 68040 has a subset of the instructions handled by the 68881/68882. Missing instructions must be emulated via the LineF exception handler. The above code blindly intercepts all LineF exceptions, including those which are now valid. Of course, removing this code diminishes MacApp’s debugging capabilities somewhat, but this should be a minor inconvenience.

MacApp and floating window support

Written: 3/11/91
Last reviewed: 7/25/91

Is there a way to add floating windows to a MacApp application?

—

A short correction, the latest MPW 2.0.1 contains experimental-and-unsupported code, and the floating window package is part of this code. Please check the source code (MacApp), and use the flag qExperimentalAndUnsupported for the build of the floating window code.

Official support for floating Windows will be available from MacApp 3.0 forward. Meanwhile the ETO CD #3 contains an unsupported version of how to implement floating windows in MacApp 2.0.

Apple hasn't set availability of MacApp 3.0; late alphas and betas will most probably be available on the ETO CD as well.

Avoiding odd-address errors on 68000-based systems

Written: 4/30/91
Last reviewed: 5/20/91

My application crashes on Macintosh SE but not Macintosh II systems while attempting to read a block of code into memory and parse it. The routine passes a pointer created by NewPermPtr to various objects, so they can extract information and increment the pointer. The MacApp debugger gives me the following message:

```
Exception #3 Address error: Word or long-word reference made to an odd address
Bad address was: INVALID! ($0005372D)
```

—

You can get odd-address errors with 68000-based machines. Starting with 68020 the CPU is able to fetch data from odd addresses. Anyway, it’s a common problem to get into trouble with data fetched from odd addresses with 68000-based platforms. Possibly the data you are saving and later retrieving has odd length, causing the next fetch to start on an odd address.

To guarantee that the data will always start on even boundaries, use the MacApp OffsetPtr global routine. Here’s the new MacApp 3.0 C++ version of this routine:

```
pascal void OffsetPtr(Ptr& p, long offset)
{
    p += offset;
    if (((long) p) & 1)
        ++p;
}
```

The function tests if the pointer is pointing at an odd address, and if so it increases the pointer one byte. You could do something similar in your member function, such as:

```
short*  sptr = (short*)buf;
fValue = *sptr;
OffsetPtr(buf,sizeof(short)); // instead of buf+= sizeof(short);
```

DAs shouldn’t be written in MacApp

Written: 4/29/91
Last reviewed: 6/21/91

Where can I find information about writing a desk accessory (DA) in MacApp?

—

First, take a look at the Macintosh Technical Note “Inside Object Pascal,” which explains why, as an application framework, MacApp should be used to write only applications and MPW tools. Apple’s online MacApp forum and the MacApp Developers’ Association (MADA) are good points of contact for exchanging information about MacApp development. Contact MacApp.Admin on AppleLink for details about the MacApp forum. MADA can be reached at 206-252-6946 or AppleLink address MADA.

MacApp segmentation
Written: 2/25/91
Last reviewed: 6/25/91

What is the recommended method of segmentation for MacApp Applications? If I pull out my object modules and test them by themselves they function correctly. My app has now grown beyond my previous segmentation scheme and I’m back to the drawing board.

—

Yes, it makes sense to plan for segmentation with MacApp, mostly because the code segments tend to become very crowded, and also because Macintosh operating system does not have virtual-page memory paging, so it’s the responsibility of the programmer to define where certain functions are in various segments. And MacApp will load those segments it needs—so the developer is in a way defining dynamically loaded libraries (and it makes sense to push certain methods which are seldom called into segments that reside most of the time on the hard disk instead of in the memory).

So try to separate all the methods in more segments, and to group methods that are related to the same segments. If possible create small segments, which load faster from disk to memory.

Here’s a suggestion list of where to place various methods in what segments (taken from the Dave Wilson MacApp training manual):

```
Segment name - routines

ARes          - routines that get called often
                  DoSetupMenus
                  DoSetCursor
                  DoIdle
                  Draw
                  DoKeyCommands, if typing

ADebug        - debugging code

AFields       - your Fields methods

AInit         - used only once (during the lifetime of the application)
                  IYourApplication

ATerminate    - used only when you quit

ASelCommand   - for selecting commands
                  DoMenuCommand
```

```
DoMouseCommand
DoKeyCommand
TYourPasteCommand
IYourCommand
```

```
ADoCommand    - for performing commands
                  TReColorCmd::DoIt
                  TSketcher::TrackMouse
                  TTypingCmd::Commit

AClipboard    - for clipboard non-commands
                  MakeViewForAlienClipboard
                  GivePasteData
                  WriteToDeskScrap

AOpen         - opening stuff
                  DoMakeViews
                  DoMakeWindows
                  IYourDocument
                  DoMakeDocument
                  IYourView

AClose        - closing stuff

AReadFile     - reading from disk
                  DoRead
                  DoInitialState

AWriteFile    - saving files
                  DoNeedDiskSpace
                  DoWrite

AFile         - rarely used

ANonRes       - for routines you rarely call

ASomething    - define your own segments and place related methods here
```

Hint: If you're unsure where a method goes, put it in same code segment as the method that calls it.

The Mouser browser has a nice feature that shows the segment in which each method is placed.

Another issue is that place segments that you don't want to be unloaded automatically from memory in the 'res!' resource (good for methods that are time critical, and should not be unloaded to disk), and use the 'mem!' resource for defining how much to add to the basic temporary memory reserve memory size, permanent memory reserve, and stack space.

MacApp 'mem!' and 'seg!' resources

Written: 3/31/92
Last reviewed: 5/21/92

What's the purpose of the MacApp 'mem!!' resources, and where does the documentation for these resources exist?

—

The 'mem!' resource allows you to change MacApp's memory allocation reserves in various ways. Each contains three numbers: the amount to add to the temporary reserve (which is used for system allocations such as system resources and temporary handles), an amount to add to the permanent reserve, which is used by you for your memory allocation, and an amount of stack space. Having multiple 'mem!' resources causes their values to be summed; in this way, you can create a "Debugging" 'mem!' resource which gives you extra space and delete it when

you produce a non-debug version. This is discussed in the MacApp 2.0 General Reference, in Chapter 3.

The 'seg!' resource is used to reserve space for code segments. If the Macintosh ever tries to load a code segment but fails due to lack of memory, it will crash. Thus, MacApp keeps a store of memory solely for loading code resources. It sizes this reserve by adding together the sizes of the segments named in the 'seg!' resource. One way to do this would be to just name all the segments, so you know there's room for them all; however, this would be wasteful, because many segments are often unused (your printing code, for example). So what you do is name only those segments that represent the largest code path you can have; the calling chain that would require the largest set of code segments to be loaded at any time. This is also described in Chapter 3 of the MacApp General Reference. In contrast, 'res!' names segments that _must_ be resident all the time; they are actually loaded and made resident, as opposed to the 'seg!' segments, which are only used to calculate how much memory should be reserved for segments in general.

InitUMacApp parameter callsToMoreMasters maximum
Written: 8/7/92
Last reviewed: 5/21/92

After testing, I believe the maximum value for the InitUMacApp parameter which specifies the number of calls to MoreMasters is 511. In nodebug mode, any number larger than that will yield one master pointer, and even that is mutated—$80 instead of $100 bytes in length. In debug mode, the application will crash immediately after it is launched. Is this limitation documented anywhere?

—

MacApp multiplies the value you provide by the current number of master pointers in a block (which is how many would normally be allocated by a single call to MoreMasters). This value is usually 64. MacApp then stores the result into the field that specifies how many master pointers to allocate in each block and then calls MoreMasters once, which allocates all the master pointers you want in one block, which is somewhat more efficient than having them in many blocks. Unfortunately, if you call InitUMacApp(512), it calculates the number of pointers as being 512*64 = 32768. You will note that the field in the zone header which holds the number of master pointers to be allocated at any time is an integer; 32768 overflows the number of bits it has available, because it is a signed number. MacApp then tells the Memory Manager to allocate one master pointer block, with a negative number of pointers in it. Needless to say, this is somewhat sub-optimal. Here are a couple of possible solutions:

1) Fix MacApp. Modify the InitUMacApp source so that if it is called with an argument greater than 512, rather than attempting to allocate all the pointers at once, it allocates more than one block. For example, the call InitUMacApp(2071) might set up the zone header so that it would allocate 64*500=32000 master pointers at once, then call MoreMasters 4 times,

then set it for 64*71=4544 master pointers at once, then call MoreMasters one last time.

2) If you don’t want to muck with the MacApp source, you can just allocate the surplus pointers yourself; modify the zone header, allocate most of your pointers, and then call InitUMacApp to allocate the rest.

This is an incredibly large number of handles; the Macintosh Memory Manager wasn’t designed for an immense number of handles in a single heap; it’s probably going to be very

slow. You may want to consider working out your own allocation scheme for some of these objects.