Technical Note PT530
MacApp Q&As

CONTENTS

Downloadables

This Technical Note contains a collection of archived Q&As relating to a specific topic--questions sent the Developer Support Center (DSC) along with answers from the DSC engineers. Current Q&A's can be found on the Macintosh Technical Q&A's web site.

[Oct 01 1990]

Extending MacApp FailOSErr call to include additional error strings

Is there any way to extend the `FailOSErr` call in MacApp to include MacTCP errors?

This is what happens: `FailOSErr` calls `Failure`, etc. Eventually `LookupErrString`(...,resourceID,...) gets called with `resourceID` = 128 -or- `errOperationsID`. `LookupErrString` then adds 1000 -or- `errAppTable`. It searches for resource 1128 of type `'errs'`. If it finds the error table resource MacApp attempts to use that resource first else it uses the MacApp supplied `'errs'` resource 128. Notice the last entry in resource 128 is defined to catch all errors in the range of -32768..32767. The first entry in the `'errs'` resource defines which `'STR#'` to use.

To handle MacTCP errors, add a `'errs'` resource ID 1128 which refers to a `'STR#'` resource.

MacApp "internal component is missing" compilation message

Date Written: 3/12/93

Last reviewed: 6/24/93

When starting a new MacApp program, I get the message "Couldn't create new document because an internal component is missing. Please contact the developer." How do I find out which component is missing?

When building a -nodebug version of an application, MABuild assumes that the code is as optimized as possible, including making the code as small as possible. Therefore the linker strips out any unused code from the final application.

The problem is that the linker determines which code is to be stripped out by finding all objects that are constructed with the "new" operator. Objects derived from TView are instantiated through calls to `TViewServer` methods instead of calling the new operator. Because of this, the linker thinks that calls to your derived View objects aren't used, so they're stripped from the nodebug build. To circumvent this problem, you have to fool the linker into not stripping out this code.

MacApp defines a macro that makes this easy for you to trick the linker. Place this line of code in your implementation of `TSomeApplication::ISomeApplication:`

MacApp takes care of this for you for any View classes defined by MacApp, but you do have to do it for any subclasses of `TView` defined in your own program. Good examples of the use of this routine can be found throughout the MacApp C++ examples, in the "{MacApp}"Examples:CPlus Examples: folder, namely in the Skeleton example.

More information about this can be found in the MacApp tutorial (C++) on page 166, and in the MacApp AppleLink archives.

MacApp hierarchical menu ID range is 1-63

Date Written: 2/24/93

Last reviewed: 4/9/93

Submenus aren't working correctly in my application. I'm using MacApp and my 'CMNU' and 'MBAR' resources are set up correctly. The submenu appears to be set up properly in the hierarchical menu bar. The main menu highlights in response to a command key for its submenu, but the submenu never shows up when selected. How can this be?

You'll see this behavior if your hierarchical menu IDs are greater than 63. MacApp requires that hierarchical menus be in the range of 1-63. For more information on menus, look at the "Working with Menus" chapter in the Programmer's Guide to MacApp.

Use MacApp 3 "-NeedsSystem7 Mabuild" flag for System 6

Date Written: 1/6/93

Last reviewed: 6/14/93

Now that I've converted my application from MacApp 2 to MacApp 3, it doesn't run under System 6, even when I don't compile for large model addressing. What switches or options do I need to observe for a System 6 application?

Your MacApp program will run under System 6 if you use MABuild's - `NoNeedsSystem7` option. The default is System 7 only. Using ModelFar for large model addressing has no effect on System 7 dependencies.

Why MacApp uses Initialize and Free methods

Date Written: 11/20/92

Last reviewed: 6/14/93

Why does MacApp use the Initialize and Free methods instead of the normal C++ constructor and destructor methods?

MacApp doesn't use constructors for historical reasons. Object Pascal was used in MacApp 2.0.1, which doesn't have constructors and destructors as part of the language, and as a result these facilities had to be provided as part of MacApp instead of the language.

MacApp 3.0 designers tried to accomplish backward compatibility with applications written with older versions of MacApp based on Object Pascal. Because of this, the designers decided to stay with the Initialize and Free functions rather than just have an instance of the objects declared and destroyed with new and free.

Which headers to use for MacApp program development

Date Written: 9/15/92

Last reviewed: 6/14/93

As a MacApp developer, am I supposed to be using the .h files in the MPW: Interfaces folder, or the ones in the {MacApp}CPlusIncludes folder?

As a default, MacApp first searches through the {MacApp}CPlusIncludes folder for header files specified in your source, so we suggest using the `CPlusIncludes` headers. The path to these include files is defined in the {MacApp}Startup Items:Startup folder.

MacApp developer technical support options

Date Written: 10/27/90

Last reviewed: 11/16/92

Is there a special address for MacApp technical support or is the Developer Support Center the correct target?

For Apple Partners and Associates, DEVSUPPORT is an option, though there are also other avenues of tech support for MacApp programmers as well. First there are the group addresses MacApp2Tech$ and MacApp3Tech$. These are groups 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 a group and receive copies of all the mail sent to it by sending a request to MacApp.Admin.

Another avenue of support is from MADA. This is a nonprofit 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) 253-2765, or write to them at AppleLink address MADA. Annual fee is $75.

Debugging options for C++

Date Written: 11/28/90

Last reviewed: 6/14/93

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

Date Written: 6/1/91

Last reviewed: 6/14/93

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, checkbox, 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

Date Written: 6/17/91

Last reviewed: 6/14/93

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:

or in MPW Pascal:

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

Date 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 a MacApp application, your implementation needs to do the following:

1. Create a record that 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}
```

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}
```

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 the window. 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);
    ...
```

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

Date 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:

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)^;
```

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 that are now valid. Of course, removing this code diminishes MacApp's debugging capabilities somewhat, but this should be a minor inconvenience.

Avoiding odd-address errors on 68000-based systems

Date 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
```

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;
```

DAs shouldn't be written in MacApp

Date 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 on-line 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) 253-2765 or AppleLink address MADA.

MacApp `'mem!'` and `'seg!'` resources

Date Written: 3/31/92

Last reviewed: 5/21/92

What's the purpose of the MacApp `'mem!'` and `'seg!'` 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 that 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 that 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're actually loaded and made resident, as opposed to the `'seg!'` segments, which are used only to calculate how much memory should be reserved for segments in general.

InitUMacApp parameter callsToMoreMasters maximum

Date Written: 8/7/92

Last reviewed: 6/14/93

After testing, I believe the maximum value for the `InitUMacApp` parameter that 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 that 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 suboptimal. 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 four 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.

Back to top

Downloadables

Acrobat version of this Note (K). Download