Technote 1173

Understanding Open Transport Asset Tracking

By Quinn "The Eskimo!"
Apple Worldwide Developer Technical Support

CONTENTS

This Technote describes how Open Transport tracks assets, like memory and providers, which it allocates on your behalf. Open Transport's approach to asset tracking was poorly documented in the past, and that contributed to a number of common developer programming errors.

This technote describes the asset tracking mechanism used for the existing Open Transport programming interface, and how that mechanism has been "tuned up" for Carbon.

This Technote is directed at all programmers who use the Open Transport client programming interface.

Goals and Problems

Open Transport regularly allocates assets on your behalf. For example, every time you open an endpoint, Open Transport creates an internal data structure to hold the state of that endpoint. Whenever Open Transport allocates an asset, it associates the asset with the particular client that allocated it, so when the client terminates, Open Transport can clean up the assets it allocated.

IMPORTANT:
In the context of this and other Open Transport technotes, ”client” refers to some program that uses the standard Open Transport programming interface, not to the client in a client/server protocol.

Open Transport tracks two important classes of assets, providers (endpoints, mappers, service providers) and memory (allocated by `OTAllocMem` and `OTAlloc`). Open Transport also has infrastructure to track deferred tasks, system tasks, timer tasks, and raw streams, although these assets are not actually tracked by the current Open Transport implementation on traditional Mac OS.

Other parts of Mac OS implement various different asset tracking schemes:

- Most assets are tracked by process. When such an asset is allocated, it is tagged as belonging to the process that was running at the time (as returned by `GetCurrentProcess`). This approach is used for assets such as windows, resource maps, file reference numbers, serial ports (when a serial port arbitrator is installed), and temporary memory.
- Other assets are tracked by heap zone. When such an asset is allocated, it is tagged as belonging to the current heap (the value returned by `GetZone`). This approach is used for assets such as code fragment connections and component instances.
- Some assets are not tracked at all. For example, the system does not track who opened a device driver (except serial ports). Consequently, the classic networking stack, which was replaced by Open Transport, did not do any asset tracking.

Unfortunately, Open Transport cannot use any of the asset tracking schemes described above. The Open Transport programming interface is typically called at deferred task time, and tracking assets by the current process or heap zone does not make sense at deferred task time.

For example, imagine that your application calls `OTAllocMem` from a deferred task. The deferred task might run while the Finder is the current application. If Open Transport tracked assets based on the current process, it would erroneously think that the Finder allocated this memory.

Because of these problems, Open Transport 1.0 introduced a new asset tracking mechanism, which is described in the next section. However, this mechanism proved somewhat confusing to developers, so Carbon introduces a much clearer mechanism, based on explicit contexts.

Back to top

The Original Solution

This section describes how Open Transport tracks assets for pre-Carbon clients, and the rules you should follow when building pre-Carbon software.

The Basics

Open Transport tracks assets for pre-Carbon clients by means of a global variable, named `__gOTClientRecord`, which is statically linked into your program’s data section (A5 world for 68K code). When you call an Open Transport asset-creation routine, you are actually calling a statically linked stub, which in turn calls a real, dynamically linked variant of the routine. This dynamically linked routine, whose name ends in “Priv”, has as ar extra parameter which is the address of the client record.

For example, when you call `OTOpenEndpoint`, you're actually calling the statically linked stub shown below.

```
// Prototype of dynamically linked routine.

extern pascal EndpointRef OTOpenEndpointPriv(
                                OTConfiguration* config,
                                OTOpenFlags oflag,
                                TEndpointInfo* info,
                                OSStatus* err,
                                OTClientRecord *client);

// Statically linked stub routine which is linked into your program.

extern pascal EndpointRef OTOpenEndpoint(
                                OTConfiguration* config,
                                OTOpenFlags oflag,
                                TEndpointInfo* info,
                                OSStatus* err)
{
    return OTOpenEndpointPriv(config, oflag, info, err, &__gOTClientRecord);
}
```

The “Priv” routine is the real entry point into the operating system, exported by the Open Transport shared libraries. Open Transport uses the address of the client record to uniquely identify the client creating the asset, an tags the asset as belonging to that client. When the client terminates (either it calls `CloseOpenTransport` or an application client terminates unexpectedly, see below), Open Transport disposes of any assets the client “leaked.

IMPORTANT:
The contents of the `__gOTClientRecord` variable are private to Open Transport. You should not modify this variable, or depend on its value.

The `__gOTClientRecord` and the statically linked stub routines are obtained from a static object file to which yo link your program. For PowerPC software, this file is an XCOFF object file whose name ends with “PPC.o”. A typical PowerPC program links with this static object file, containing the client record and the stub routines (for application clients, this is “OpenTransportAppPPC.o”), and a CFM stub library (“OpenTransportLib”), which references the dynamically linked asset-creation “Priv” routines and other, non-asset creation, routines.

IMPORTANT:
For more information about linking with Open Transport libraries, see DTS Q&A NW 18 Open Transport Libraries.

Applications versus Extensions

To further complicate matters, Open Transport has two sets of static object files. The set whose name contains “App” is only suitable for applications, while the set whose name contains “Extn” is suitable for other, non-application code (system extensions, shared libraries, MPW tools, and so on). Your choice of libraries affect the following.

- Automatic Cleanup—If you link to the “App” libraries, Open Transport will patch `_ExitToShell` to automatically call `CloseOpenTransport` for you (if you haven't) when you application quits.
- Client Pool—If you link with the “App” libraries, your client pool is located in the application zone. If you link with the “Extn” libraries, your client pool is located in the system zone. See DTS Technote 1128 Understanding Open Transport Memory Management for details on client pools.

Rules for Clients

Open Transport's asset tracking strategy has a number of important consequences for developers.

- You must call `InitOpenTransport` once (and only once) for each global data section (A5 world for 68K) you have. If you have an application and a shared library, both of which need to create endpoints, you must call `InitOpenTransport` once for the application and once for the shared library.
- If you are writing non-application code, you must remember to call `CloseOpenTransport` before your global data section unloads. If you're writing an application, Open Transport's `_ExitToShell` patch should clean up automatically for you, however DTS recommends that your application not rely on this safety net in the typical case. In general, you should call `CloseOpenTransport` once for each time you'v called `InitOpenTransport`.
- If you are writing a 68K code resource that calls Open Transport, it is vital that you have a valid A5 (or global) world. This is because the Open Transport client record is a global variable allocated in your cod resource's A5 world. The Open Transport libraries (specifically "OpenTransport.o" and "OpenTransportExtn.o") reference this variable using A5-relative addresses. An A4 "global world" (a used by Metrowerks and Symantec compilers) is not sufficient.

IMPORTANT:
If you are using the Metrowerks CodeWarrior environment to build a code resource that calls Open Transport, you should base it on the OTCodeResource sample, which shows how to use Open Transport's A5-based libraries in an A4-based code resource. It is important that you use version 1.0.1b1 (or later) of this sample. Earlier versions had a crashing bug.

- A code resource cannot "borrow" the A5 world from its host application when calling Open Transport. This technique, which is commonly used by plug-ins that need to call QuickDraw, is not possible for Open Transport clients because there's no guarantee that the host application has an Open Transport client record. Even if it does, there's no way to find the offset of the client record in the host's data section.
- If you transfer providers from one Open Transport client to another (from a shared library to your application, say, or between applications), you must call `OTTransferProviderOwnership` to ensure that Open Transport knows that the provider now belongs to the new client. Otherwise, when the previous owner calls `CloseOpenTransport`, Open Transport will clean up the provider from underneath you.
- Any non-application code should use the "Extn" libraries. When deciding which libraries to link with, you should ask yourself "Do I want my connection to Open Transport to be cleaned up automatically when `_ExitToShell` is called?" and "Are my global variables guaranteed to be around when `_ExitToShell` is called?" If both answers are "Yes," use the "App" libraries, otherwise use the "Extn" libraries.
- When debugging Open Transport programs in MacsBug, you will not be able to find symbols for the stub routines. For example, you won't be able to put a transition vector break on "InitOpenTransport"; you should put it on "InitOpenTransportPriv" instead.

Tracked Assets

The follow table shows exactly which routines are have "Priv" variants. Except as noted below, the assets create by these routines are tracked by Open Transport and will be disposed of when the client calls `CloseOpenTransport`.

Initialization and Termination	`InitOpenTransport`
	`CloseOpenTransport`
	`OTRegisterAsClient`
	`OTUnregisterAsClient`
	`OTTransferOwnership`

Memory	`OTAllocMem`

	`OTAlloc`
	`OTFreeMem (1)`
	`t_alloc`

Provider Open and Close	`OTOpenEndpoint`
	`OTAsyncOpenEndpoint`
	`OTOpenAppleTalkServices`
	`OTAsyncOpenAppleTalkServices`
	`OTOpenInternetServices`
	`OTAsyncOpenInternetServices`
	`OTOpenMapper`
	`OTAsyncOpenMapper`
	`OTOpenProvider`
	`OTAsyncOpenProvider`
	`OTCloseProvider`
	`t_open`
	`t_close`

Raw Stream Open and Close	`OTStreamOpen (2)`
	`OTAsyncStreamOpen (2)`
	`OTOpenEndpointOnStream (2)`
	`OTOpenProviderOnStream (2)`
	`OTStreamPipe (2)`
	`stream_open (2)`
	`stream_pipe (2)`

Task Creation	`OTCreateDeferredTask (2)`
	`OTCreateSystemTask (2)`

Notes:

1. `OTFreeMemPriv` is only exported for the benefit of clients linked with the Open Transport 1.1 and earlie libraries. Clients linked with the Open Transport 1.1.1 and higher libraries will call directly into a dynamically linked version of `OTFreeMem` which automatically works out which client allocated the memory.
2. These routines have "Priv" variants so that a future version of Open Transport can track their assets. Th assets created by these routines are not tracked by current versions of Open Transport.

Back to top

The Carbon Solution

The original solution to the Open Transport asset tracking problem was creative, but it has a number of problems.

- Lack of Understanding—The original solution was poorly documented and therefore badly understood, which resulted in many asset tracking related crashes. For example, linking a code resource with the "App" libraries will cause a crash when the host application quits, as Open Transport's `_ExitToShell` patch tries to execute code that has been unloaded.
- Linker Semantics—Mac OS programmers are not used to providing semantic content through their choice of linker libraries. Any such approach is inherently error prone.
- Static Object Files—Alternative development environment (Java runtimes, BASIC implementations, Forth interpreters, C++ cross compilers, and so on) have easy mechanisms to provide access to CFM libraries. However, Open Transport's static object files—in MPW `'OBJ '` format for 68K or XCOFF format for PowerPC— present serious difficulties for anyone using these alternative environments.

To address these problems—especially the problems with static object files—the designers of Carbon chose to restructure the Open Transport programming interface for Carbon programs. This minor update, which is in line with the general “tune up” goal of Carbon, is described in the next section.

The Basics

The Open Transport programming interface for Carbon programs makes the client record an explicit parameter to the routines which need it. For example, rather than call `OTOpenEndpoint`, in Carbon you must call `OTOpenEndpointInContext`. This routine takes an addition parameter of type `OTClientContextPtr`, which represents the client which is opening the endpoint.

Your program must create an Open Transport context explicitly by calling `InitOpenTransportInContext`. This routine, whose prototype is shown below, takes two parameters. You use the first parameter to explicitly tell Open Transport whether you are an application or extension client. The second parameter returns the client context pointer, which you must pass to other asset-creation routines.

```
extern pascal OSStatus InitOpenTransportInContext(
                            OTInitializationFlags flags,
                            OTClientContextPtr *outClientContext);
```

Carbon also includes the concept of a default application context. If you pass `nil` to the `outClientContext` parameter of `InitOpenTransportInContext`, you initialize this default application context. From there on, you can pass `nil` to any “InContext” routine to tell it to act in this default context.

Finally, to help application developers maintain source code compatibility between their Carbon and pre-Carbon source bases, the Carbon designers introduced a set of macros that map the pre-Carbon routines to their Carbon equivalents. For example, the following macro allows you to open an endpoint under Carbon without changing your pre-Carbon source.

```
#define OTOpenEndpoint(config, oflag, info, err) \
        OTOpenEndpointInContext(config, oflag, info, err, NULL)
```

To access these macros, you must define the `OTCARBONAPPLICATION` compile-time variable. These macros are only suitable for application developers because they always use the default application context. Shared libraries must always use the “InContext” routines.

Tracked Assets In Carbon

The follow table shows exactly which routines are “InContext” under Carbon. Except as noted below, the assets created by these routines are tracked by Open Transport and will be disposed of when the client calls `CloseOpenTransportInContext`. This list is significantly shorter than the list of “Priv” routines given earlier because Carbon does not support many low-level Open Transport routines.

Initialization and Termination	`InitOpenTransportInContext` `CloseOpenTransportInContext`
Memory	`OTAllocMemInContext` `OTAllocInContext`

Provider Open and Close	`OTOpenEndpointInContext`
	`OTAsyncOpenEndpointInContext`
	`OTOpenMapperInContext`
	`OTAsyncOpenMapperInContext`
	`OTOpenAppleTalkServicesInContext`
	`OTAsyncOpenAppleTalkServicesInContext`

	`OTOpenInternetServicesInContext` `OTAsyncOpenInternetServicesInContext`

Task Creation	`OTCreateDeferredTaskInContext (1)`
	`OTCreateTimerTaskInContext (1)`

Notes:

1. These assets are tracked by the Mac OS X implementation of the Open Transport programming interface, but are not tracked by current versions of Open Transport on traditional Mac OS.

Back to top

Summary

Understanding how Open Transport tracks assets is important when writing applications, shared libraries, code resources, and other software that calls Open Transport reliably. The most important rule for pre-Carbon clients is that you must call `InitOpenTransport` and `CloseOpenTransport` once per global data section. The Carbon programming interface to Open Transport makes this requirement explicit and simplifies the Open Transport asset tracking story.

Further References

- DTS Q&A NW 18 Open Transport Libraries
- DTS Technote 1128 Understanding Open Transport Memory Management
- Inside Macintosh: Networking with Open Transport
- DTS Sample Code OTCodeResource
- Carbon Specification
- DTS Q&A NW 36 Calling CloseOpenTransport When Writing an Application

Back to top

Downloadables

Acrobat version of this Note (how many K?)

Back to top

Acknowledgments

Thanks to Joe Holt, Rich Kubota, and Vincent Lubet.

To contact us, please use the Contact Us page.
Updated: 23-August-1999

Technotes | Contents
Previous Technote