TECHNOTE 1095

Object Support Library Version History

By Andy Bachorski
Apple Developer Technical Support (DTS)
devsupport@apple.com

CONTENTS

This Technote is an attempt to clarify the version history of Apple's Object Support Library (OSL). This library provides routines that applications can use to support the Open Scripting Architecture (OSA) object model.

OSL was originally released as a 68K static library. With the introduction of Power Macintosh systems, OSL was repackaged as a shared library. When the Code Fragment Manager 68K Runtime Enabler (CFM-68K) was released, it became a fat library containing both Power PC and CFM-68K versions of OSL.

This Technote lists all versions of OSL that are currently available along with a brief history and description of each one, and recommendations as to which versions to use (and not use).

If you develop Macintosh applications that provide OSA object model support, or need to use applications that do, you should read this Technote.

In the Beginning of OSL

When the OSL was first released, it was a 68K static library (`AEObjectSupportLib.o`) that was statically linked into applications that supported the OSA object model. This is still the case for classic 68K applications.

With the release of the Power Macintosh and the accompanying CFM shared library application model, it was decide that a shared library version of the OSL (`ObjectSupportLib`), rather than a '.o' file, should be provided for PowerPC-native applications. This would allow native applications to take advantage of the shared library application model.

The OSL PowerPC shared library was released as version 1.0.2. It was included with the first Power Macintosh system software (version 7.1.2) and the AppleScript SDK version 1.1. It is currently still available on the Mac OS SDK CDs. This version of the OSL has a number of known bugs in its handling of `whose` clauses. (These bugs are listed in the Known Bugs in the OSL section of this Technote.)

This first shared library introduced the first OSL shared library problem. The OSL was written in Pascal, but there were no PowerPC Pascal compilers, so it contained a small PowerPC-native library that loaded a 68K code resource containing the OSL.

But this loading was not done properly; the `ObjectSupportLib`'s resource file often ended up in the middle of an application's resource chain. This problem was worked around in system 7.5.2 by forcing the Finder to load the OSL when it started up, thus forcing OSL's resource file to be located in the system, where it did no harm. This work-around masked the problem for all system versions after 7.5.2, but the problem is still there for earlier system versions.

After the introduction of the Power Macintosh, work began to convert the OSL source code from Pascal to C. This conversion was needed in order to provide a shared library containing both PowerPC and CFM-68K code. The converted fat OSL was released as version 1.0.4, which was included on the E.T.O. CD-ROM releases until recently.

There were a number of problems, however, with this version of the OSL, the worst of which was that a Gestalt selector installed by version 1.0.2 was not being installed by version 1.0.4. This prevented applications that were testing for this Gestalt selector from detecting the presence of the OSL shared library.

Note: A Gestalt selector is not the best method for detecting the presence of the OSL, but this decision was made early in the development of the code fragment model and the liabilities of using Gestalt were not yet well understood. The preferred method would have been to have developers compare a symbol in the library to `kUnresolvedCFragSymbolAddress`. (For details of this process, see Technote 1083: Weak-Linking to a Code Fragment Manager-based Shared Library)

The other major problem with this version of the OSL was a number of bugs in the code that handled `whose` clause resolution (in addition to those previously mentioned). These bugs would cause an application to return incorrect results and/or error messages when they were presented with valid requests. Because of the missing Gestalt selector which prevented this version's use, the `whose` clause resolution bugs were not discovered until much later.

No Code Changes, Just Version Changes

Somewhere between versions 1.0.2 and 1.0.4, a mistake was made in the build process for the AppleScript 1.1 SDK which led to version 1.0.2 of the OSL being released as version 1.1. There is no difference in the actual code, only a new version number.

This meant we now had a 1.1 version which is really older than version 1.0.4. But installers complain when you try to install v1.0.4 over 1.1 because they think you are replacing a newer version with an older one. So when Apple shipped the Apple Telecom software version of OSL, 1.0.4 was changed to 1.1.1. This allowed the installer to replace the old version with the new. Again, no code changes, just a version change. (All the old problems, of course, were still there).

This is the point where many people started having problems; we now had a situation where a (generally) well-behaved version 1.1 was being replaced with a broken version 1.1.1.

A Valiant Attempt To Fix Things

The AppleScript team, at this point, decided that enough was enough -- it was time to move forward and fix things. They released a version that fixed the Gestalt selector problem and called it 1.1.1. But no one had told them about the Apple Telecom OSL version 1.1.1, so they changed the version to 1.1.2. But, again, someone else had already made a limited release of an OSL version 1.1.2.

This was unfortunate, but not a major problem -- they just incremented the version to 1.1.3. This version, which contained the fixed Gestalt selector, got a limited developer release. Great -- now the applications which couldn't load it before due to the Gestalt selector bug could use the OSL. But when these applications loaded the OSL, they revealed all sorts of interesting behaviors resulting from the OSL's inability to resolve `whose` clauses. (Remember that no one had actually executed this code before the Gestalt bug was fixed.)

Back to the Drawing Board

The AppleScript team started looking at the 1.0.4/1.1.1 code stream in more detail and, after considering what a fix might mean to the stability of existing applications, it was decided that the risk of further trouble was too great. The decision was made to return to the older 1.0.2/1.1 code stream for the next release of the OSL.

As a result, the resource-loading source stream in version 1.0.2 was revised so it could be compiled as a native PowerPC and CFM-68K library. The resource-loading bug was also addressed. This revised library was tested, sent to a small group of developers for further testing, and then approved for final build as version 1.1.4.

Along the way, though, there was a new wrinkle added to this story. In order to fit in with the Mac OS reference release strategy, the file's creator type got changed, along with with its version number, which changed to 1.1.6.

Note:
Version 1.1.4 appears on the Developer CD Series and the Mac OS SDK CD, while version 1.1.6 appears on certain E.T.O. CDs.

But we still have more to our story. It seems that the test suite used to validate the OSL tested for the presence of the Gestalt selector, but did not determine if it was properly installed. Basically, native OSL resource-loading code wasn't doing the right thing when it installed its Gestalt selector, which could lead to crashes in certain situations.

In the meantime, Version 1.1.6 was included as part of the Harmony (Mac OS 7.6) f3 build that was seeded to developers. When this latest OSL problem was discovered, the decision was made to take the version 1.0.2 of OSL and reversion it (again) as version 1.1.8. This is the version of OSL that is included in the GM version of Mac OS 7.6.

The biggest problem with this version is that it isn't fat, as it doesn't contains CFM-68K code (recall that 1.0.2 didn't, either). This wasn't an immediate problem, since Mac OS 7.6 shipped without support for CFM-68K. In fact, 7.6 explicitly checks for the presence of the current CFM-68K extension and disables it to prevent its loading with this version of the system. If you can't run CFM-68K, you don't need a fat OSL.

A New Hope

Thanks to the efforts of a number of engineers, the OSL got a more thorough rewrite which fixed both the Gestalt selector and resource file bugs. The testing was completed and the problems were resolved. This time, the OSL was released as version 1.2, which now generally works as expected. It still contains the bugs related to `whose` clause resolution, but they are not fatal and are easy for developers to work around (see Known Bugs in the OSL).

The World As We Know It

With all this said and done, you're probably scratching your head and wondering, "What version should I be using, anyway?" At this time (April, 1997) the answer to that question is (drum roll, please):

You should be using version 1.2 of the Object Support Library.

If you don't have access to this version, then use 1.1.8 (or one of its twins, versions 1.0.2 or 1.1). **Keep in mind that these older versions are not fat**, which means that they won't work with CFM-68K applications. **If you need to run a CFM-68K application, you must use version 1.2**.

And, of course, you should install newer version of the OSL as they are released.

The History of OSL

Version	Status
1.0.2	First PowerPC shared library, resource file bug
1.0.4	New code base, no Gestalt selector, other bugs
1.1	Really just 1.0.2 in disguise, new creator code
1.1.1	1.0.4's twin (warts and all)
1.1.2/1.1.3	Never publicly released, so don't look for it
1.1.4	Doesn't work, but released on OS SDK CD
1.1.6	Version that shipped with Harmony f3
1.1.8	Version that shipped with Mac OS 7.6 (another 1.0.2 clone)
1.2	Shipped with CFM-68K 4.0; fixes all known crashing bugs

Known Bugs in the OSL

There are still several known bugs in the OSL, but they all have workarounds.

Unlocked Handles Passed To Compare Functions

You can install an object comparison function for use when resolving `whose` clauses. When the OSL calls your comparison function, it passes pointers to two `AEDescs`: one for the object being compared, and one for the object or descriptor to compare to the first.

The problem here is that the memory blocks containing the descriptor records pointed to by the parameters are located in relocatable blocks that are not locked and may move after the compare function is called.

The workaround for this problem is to copy the descriptor records pointed to by the parameters to a local variable as soon as you enter the object comparison routine.

There is a complication to this problem involving Classic 68K applications and the segment loader. If the comparison function and the `AEObjectSupportLib.o` library are not in the same segment when you build your application, it's possible for the descriptor records to move before they are copied if the segment containing the OSL is not already loaded.

The workaround for this problem is to make sure that the comparison function and the `AEObjectSupportLib.o` library are in the same segment when you build your application.

Memory Leak in Object Comparison Callback Function During `whose` Clause Resolution

There is a memory leak when the OSL calls an application-supplied comparison callback function while resolving `whose` clauses in object specifiers. When the OSL passes objects to the comparison callback function, those objects will often be application-defined tokens that are created by object accessor functions.

The problem is that the OSL calls `AEDisposeDesc` on these token objects rather than `AEDisposeToken`, which causes a memory leak -- any data the application has attached to the token is not properly disposed of.

This problem is further complicated by the unlocked handles bug described above. The workaround is to combine the unlocked handles fix with disposing the tokens yourself. The pseudo-code below describes the work-around for this problem and also the unlocked handle problem above:

Pascal version

```
function  MyCompareObjects( comparisonOperator:             DescType;
                            (CONST) VAR theObject:          AEDesc;
                            (CONST) VAR objOrDescToCompare: AEDesc;
                            VAR compareResult:              boolean): OSErr;
var
    theObjectCopy             : AEDesc;
    objOrDescToCompareCopy    : AEDesc;

begin
    { First make copies of the descriptors because the OSL has them pointing
      into a relocatable block, which can be moved by the code below. }
    theObjectCopy := theObject;
    objOrDescToCompareCopy := objOrDescToCompare;

    { Now set the original desciptors to the null descriptor since we have
      to dispose of the objects below because of this OSL bug! }
    SetToNullDesc( theObject );
    SetToNullDesc( objOrDescToCompare );

    { Code to do the comparison goes here }
    MyCompareObjects := DoTheComparison( comparisonOperator, theObjectCopy,
                                         objOrDescToCompareCopy, compareResult );

    { These descriptors are supposed to be const, but the OSL never calls
      our dispose token callback function, so we dispose of them here in
      case one of them is an application-defined token. }
    MyDisposeToken( theObjectCopy );
    MyDisposeToken( objOrDescToCompareCopy );
end;
```

C version

```
OSErr MyCompareObjects( DescType        comparisonOperator,
                        const AEDesc    *theObject,
                        const AEDesc    *objOrDescToCompare,
                        Boolean         *compareResult )
{
    AEDesc  theObjectCopy;
    AEDesc  objOrDescToCompareCopy;
    OSErr   anErr;

    theObjectCopy = theObject;
    objOrDescToCompareCopy = objOrDescToCompare;

    SetToNullDesc( theObject );
    SetToNullDesc( objOrDescToCompare );

    anErr = DoTheComparison( comparisonOperator, theObjectCopy,
                             objOrDescToCompareCopy, compareResult );

    MyDisposeToken( theObjectCopy );
    MyDisposeToken( objOrDescToCompareCopy );
}
```

Memory Leak in `marking` Callback Functions During `whose` Clause Resolution

There is a memory leak when the OSL calls application-provided marking functions during the processing of `whose` clauses in object specifiers.

In the process of resolving an object specifier, a descriptor record is created and passed to the `mark token` callback function (as the `containerToken` parameter) and object-marking callback function (as the `theToken` parameter). The problem is that the OSL never disposes of this descriptor, either through `AEDisposeToken` or `AEDisposeDesc`.

The problem can be avoided by having your application resolve all `whose` clauses it receives on its own. This way it can avoid calling `AEResolve`. This can be advantageous, especially for the most common types of object specifiers. However, you can potentially receive "uncommon" object specifiers that are better handled by the OSL using your callback functions; if you handled them yourself you'd end up duplicating most of the OSL.

Fortunately, the workaround for this problem is relatively simple: dispose of the descriptor record at the end of the object-marking function and set it to a null descriptor. This fix should not cause problems if this bug is fixed in a future release of the OSL, since it's always safe to dispose of a null descriptor record.

Function to set descriptor record to null descriptor

```
void SetToNullDesc( AEDesc theObject )
{
        theObject.descriptorType = typeNull;
        theObject.dataHandle = nil;
}
```

Further Reference

- Technote 1083: Weak-Linking to a Code Fragment Manager-based Shared Library

Acknowledgments

Special thanks to Chris Espinosa, Roger Pantos, Bryn Oh, and Otto Schlosser.

Send feedback to devsupport@apple.com
Updated: 31-March-97