ASLM 2.0 Developer’s Guide Supplement

Introduction

This document provides information about ASLM 2.0 for PowerPC and 68k. It supplements the existing documentation for ASLM 1.1.1 in the ASLM Developer’s Guide and provides information about differences between the 68k and PowerPC versions ASLM. It also documents known bugs, errors and ommisions in the ASLM Developer's Guide, and bug fixes and enhancements made since ASLM 1.1.1.

The major new feature in ASLM 2.0 is support for native PowerPC shared library and client development. Other changes since ASLM 1.1.1 mostly consist of bug fixes and minor enhancements. The sections that follow describe everything that is new in ASLM 2.0.

Paragraphs marked with a single width change bar are for ASLM 2.0b6 and 2.0b7 and paragraphs marked with a double width change bar are for ASLM 2.0b8 through 2.0f3.

New Items in the ASLM Folder

There are many new files in the ASLM folder, most of which are there to support the PowerPC. Files that end with “PPC” are simply PowerPC versions of pre-existing 68k applications and libraries. The exceptions to this are LibraryBuilderPPC and CreateLibraryLoadRsrcPPC. They are actually 68k MPW tools used to build PowerPC shared libraries.

The “ASLM Developer Tools:Libraries:” folder has been reorganized and contains some new files. The libraries for THINK and Symantec C++ have been moved to this folder, a new asubfolder has been created to support MrC, and the libraries that support CFront have been moved to a subfolder. The organization is now as follows:

•THINKLibraries is for users of the Symantec THINK development environment.

•SCLibraries is for users of the Symantec SC and SCpp MPW tools.

•CFrontLibraries is for users of the C and CFront MPW tools.

•MrCLibraries is for users of the MrC and MrCpp MPW tools.

ASLM Support For PowerPC

ASLM 2.0 provides support for 68k and native PowerPC shared libraries and clients. Emulated shared libraries and clients are supported on the PowerPC by ASLM 2.0 for 68k, which is the same as ASLM 1.1.2 with a few bug fixes and minor enhancements added. ASLM 1.1.2 also supports emulated shared libraries and clients on the PowerPC.

ASLM for 68k provides runtime support in the Shared Library Manager extension. ASLM for PowerPC provides its runtime support in a separate extension named Shared Library Manager PPC.

Mixed Mode Support

ASLM 2.0 does not provide a mixed mode solution so it is not possible for a native client to call an emulated shared library or an emulated client to call a native shared library unless the developer provides the mixed mode glue.

ASLM “on top of” CFM

ASLM for PowerPC does use the Code Fragment Manager (CFM), but only because all native executables on the PowerPC are linked as code fragments. ASLM does not use CFM to export any functions or C++ methods. The mechanism used by ASLM for PowerPC to handle dynamic linking is the same as the mechanism used by ASLM for 68k. The only difference is that the stubs linked with the client are in PowerPC assembler instead of 68k assembler and the stubs call exported functions through descriptor records (also called a DS records or transition vectors).

Since ASLM PowerPC shared libraries are code fragments, they may import functions and data from other code fragments using CFM. However, they may not use CFM to export any data or functions.

Development Environment Support

ASLM for PowerPC only supports Metrowerks and the MPW based Macintosh on Risk development tools (MrC, MrCpp, PPCLink, and friends). PPCC is not supported.

Note: Throughout this document, references to "MrC" include both the MrC and MrCpp compilers.

The Metrowerks CodeWarrior development environment is only supported for the development of clients. You cannot build a shared library using CodeWarrior. This is similar to the current support for users of the Symantec 68k THINK development environmnet. You will need to import the LibraryManagerPPC.o file from the MrCLibraries folder. You may also need to import the MPW PPCCRuntime.o library. In addition you will also need to supply the implementation of a small routine called `ASLMatexit()` as follows:

```
#ifdef __cplusplus
  extern "C" int ASLMatexit(Ptr foo);
#endif
```

```
int ASLMatexit(Ptr foo)
{
    return(1);
}
```

Be sure to link the routine in before LibraryManagerPPC.o.

There are no plans to support the RS6000 development environment.

The ASLM header files and examples all require the use of the MPW Universal Includes for both the 68k and PowerPC development. Is is also recommended that you use the version of MPW from the same ETO CD that the ASLM release is on.

68k ASLM features that are not supported

There is no support for unloading individual shared library code segments since CFM code fragments are always all in one code segment.

Global World routines

A number of changes were made to get rid of some of ASLM's requirements that on the A5 world for PowerPC clients. Applications still need to have a minimal A5 world and ASLM still uses CurrentA5, however the A5 world is no longer used or setup for `TNotifiers`, `TOperations`, and `TSchedulers` created and used by PowerPC clients. The changes listed below are all related to this. It is also recommended that PowerPC users no longer use global world routines like `OpenGlobalWorld()` and `SetCurrentGlobalWorld()`. They usually will serve no purpose and will no longer have the desired affect when creating `TOperations` and `TNotifiers.` Instead, you should set the current client to the client that you want to be saved along with the `TOperation` or `TNotifier`. 68k users should still continue to set the global world before creating `TOperations` and `TNotifiers` if they want a certain current global world saved along with the `TOperation` or `TNotifier`.

The `TNotifier` constructor has been changed for PowerPC users so that it saves the current client at the time that the notifier was constructed instead of saving the current global world.

The `TNofier::Notify()` method will only restore the client for PowerPC users. It will not restore the global world.

The `TScheduler` constructor has been changed for PowerPC users so that it saves the current client at the time that the scheduler was constructed instead of saving the current global world.

`TScheduler::IsSchedulerWorldValid()`, `GetSchedulerWorld()`, and `SetSchedulerWorld()` are not available to PowerPC users. Use the new client routines instead.

The `TOperation` constructor has been changed so for PowerPC users so that it saves the current client at the time that the operation was constructed instead of saving the current global world.

`TOperation::GetSchedulerWorld()`, and `SetSchedulerWorld()` are no longer available. Use the new client routines instead.

`SetClientToWorld()` and `GetClientFromWorld()` are not available to PowerPC users.

Building a PowerPC Shared Library

A few changes have been made to the shared library build process to support building PowerPC shared libraries.

• The BuildSharedLibrary and LinkSharedLibrary scripts support both 68k and PowerPC shared libraries, but you must specify the `-powerpc` option to build a PowerPC shared library.

•BuildSharedLibrary and LinkSharedLibrary assume that you are using PPCLink 1.2 or later. If you are using an older version of PPCLink then you will need to pass the -oldPPCLink option to BuildSharedLibrary and LinkSharedLibrary.

• There are two new MPW tools to support building PowerPC shared libraries: LibraryBuilderPPC and CreateLibraryLoadRsrcPPC. They provide the same functionality as their 68k versions, LibraryBuilder and CreateLibraryLoadRsrc, and are used automatically by BuildSharedLibrary and LinkSharedLibrary when the `-powerpc` option is specified.

• The `-makepef` option was added to BuildSharedLibrary and LinkSharedLibrary to allow options to be passed on to MakePef. This option should be followed by a quoted string which will be passed on to MakePef.

• The `-makesym` option was added to BuildSharedLibrary and LinkSharedLibrary to allow options to be passed on to MakeSym. This option should be followed by a quoted string which will be passed on to MakeSym. If `-makesym` is not specifed then the default options are “`-w -r`”.

• The `-xcoffSymFile` option was added to BuildSharedLibrary and LinkSharedLibrary to allow the shared library’s XCoff file to be used as the symbol file rather than using MakeSym to generate the symbol file.

• The `-model`, `-macsbug`, `-near`, `-privatenear`, `-nomerge`, and `-fixp` options are not supported for PowerPC shared libraries.

• The `-clientFile` and `-privateClientFile` options should be used for specifying the client object files, although `-far` and `-privatefar` will also work.

• The ‘libr’ resource has been renamed to ‘Libr’ for PowerPC shared libraries and ‘libi’ has been renamed to ‘Libi’. This allows you to create “fat” shared library files by providing both 68k and PowerPC versions of the shared library in the same file. You only need to make sure that you give the shared libraries unique code resource types.

• ASLM for 68k provides a statically linked file called LibraryManager.o. For PowerPC builds, LibraryManagerPPC.o takes the place of LibraryManager.o and LibraryManagerPPC.debug.o takes the place of LibraryManager.debug.o.

• If you are exporting a C++ class and you want to set the class's `newobject` flag, or if you are using the `clientdata` option and specify a data structure, BuildSharedLibrary requires that you build your input object file with `-sym on` so it can find the necessary size information about the class you are exporting. This was also true of ASLM 1.1.1. However, with the MrC compiler this causes a problem because you cannot build with `-sym on` and also get compiler optimizations. The solution to this is to separately compile any header file containing a C++ class that you want to set the `newobject` flag for, and use PPCLink to combine the result with the file you pass to BuildSharedLibrary as the input object file. You will need to use the MrC `-dialect cplus` and `-sym on,alltypes` options to get the header files to compile properly. The result will be an object file that contains all the symbols in the header file and no code.

Support for Non-application Code Fragments

If you have a PowerPC code fragment that is not the application's main code fragment, then you need to call `FragmentIsNonApplicationASLMClient()` before calling `InitLibraryManager()`.

Static Object Support

ASLM supports automatic static object initialization in shared libraries.

Library, Class, and Function Set id's

If you are building PowerPC shared libraries that also have 68k versions, make sure that you use different library, class, and function set id's for the PowerPC and 68k versions. Currently this is not important because information about 68k and PowerPC shared libraries is maintained in separate data bases. However, in the future this may not be true, in which case one of the shared libraires would "hide" the other and cause a crash. For example, if the PowerPC library was hidden and a PowerPC client ended up trying to use the 68k shared library, it would crash because the mixed-mode switching would not be supported/implemented.

Debugging

ASLM supports source level debugging using the "Power Mac Debugger" (a.k.a. R2DB and Macintosh Debugger for PowerPC) source level debugger.

Changes to the Examples and Test Tools

Only three of the examples in the ASLM Examples folder have been converted to PowerPC: Inspector, TestTools, and ExampleLibrary. All makefiles ending in .CFront or .SC are used to make the 68k version of the example and all makefiles ending in .MrC are used to make the PowerPC version of the example. The objects are placed in separate subfolders of the :Objects: folder, depending on which version is built. Likewise, all binaries are placed in separate subfolders of the Built folder .

• The Inspector example builds the InspectorPPC, InspectorLibraryPPC, and WindowStackerLibraryPPC binaries.

• The TestTools example only builds the TestLibraryPPC shared library. Since there is currently no support for native MPW tools (in a GM version of MPW), there is no native version of the TestTool MPW tool. However, most of the functionality of TestTool has been placed in the Tests menu of the Inspector.

• The ExampleLibrary example builds the ExampleLibraryPPC shared library and the TestExampleClassPPC application which contains some of the functionality of the LibraryMgrTest1 MPW tool.

The PowerPC version of the TraceMonitor is called TraceMonitorPPC.

Bug fixes since ASLM 1.1.1

Fixed a bug in the ShutDownActionAtom used by the installer script. The bug caused ASLM to continue to track library files moving into and out of the extensions folder while the Installer was running. This normally did not cause any problems. This bug was also reported to have caused Installer 4.0 to crash. This bug was also fixed in ASLM 1.1.2.

Fixed a bug with `NewObject()` not working with classes that use multiple inheritance.

Fixed LibraryBuilder `#if` handling so that if an outer `#if` evaluates to `0`, all inner `#ifs` and `#elses` evaluate to `0` also.

Made the ASLM Installer no longer install the INIT(22) resource under System 7. The INIT was causing some problems. However, not installing the INIT removes the fix that made ASLM still load when AppleTalk is turned off and System 7.0.1• is installed. Users will need to upgrade to System 7.1 or later or turn AppleTalk on to work around this problem. Note that this change was already made in the installer scripts on the ASLM 1.1.1 Licensing disk.

Renumbered the 68k AINI resource from 1 to -32700 and the PPC AINI from 3 to -32702 and made the installer scripts remove AINI(1) and AINI(3). This was done to accommodate certain system software services that required ASLM to be loaded slightly earlier than in was being loaded. This will break MacSNMP 1.0.x. by causing the AppleTalk Transport Agent to crash at boot time. MacSNMP 1.1 fixes this problem.

If `InitLibraryManager()` is called twice by a client (without calling `CleanupLibraryManager()` in between), it simply returns right away without doing anything. `kNoError` is returned, but in the debug version you'll go into macsbug with a message.

Fix potential bug with `RegisterLibraryFileFolder()` and `UnregisterLibraryFileFolder()` trashing the folder name passed to it if the folder name was actually a path name.

Made sure that all `Preflight()` calls are cleaned up when a client calls `CleanupLibraryManager()`, a client does an `ExitToShell`, or when a library is unloaded.

Fixed a problem with forcing libraries to unload when they have dependencies on each other. Libraries are forced to unload when you do an "Unload ASLM" or "Reload ASLM" from the Inspector and when you unregister a library file or folder and pass `true` for the `forceUnload` parameter.

Fixed a problem with BuildSharedLibrary not always generating the proper vtable initialization code for classes compiled with SCpp. The last virtual function of each class in the inheritance would not be set properly unless it was overridden.

Made BuildSharedLibrary always warn if a function set ID does not have a version number rather than only giving the warning if the function set exports a function by name.

You will no longer be in system mode when your shared library's InitProc and CleanupProc are called or when your static objects are constructed and destructed. You can still rely on the system heap being the current heap and the shared library being the current client.

In LibraryManager.h, removed the `#undef NULL` that was just before the `#include <string.h>`. The MetroWerks compiler didn't like this and it really isn't necessary anymore.

In LibraryManagerUtilities.h, got rid of the function declarations that appeared after inline assembly implementations. Neither Symantec or MetroWerks like this and its only purpose was to help make the declaration easier to read.

You will now only get a debugger break for duplicate library id's at boot time, otherwise the warning is only sent out to the TraceMonitor. This way if you have a loaded shared library that you want to replace and then reboot, you don't get the warning if you drag the in-use shared library to the trash and then install the replacement.

Fixed a problem that caused vtables to have `_pure_virtual_called()` in entries that should have had inherited method pointers. This problem would occur if 3 or more classes were involved in an inheritance chain, 2 of the classes were in the same shared library, one of the classes in the middle of the inheritance chain was in a separate shared library, the library containing the class in the middle of the chain was loaded first, and later you tried to create an instance of the class at the bottom of the chain. For example, if A inherits from B which inherits from C, A and C are in Library1 and B is in Library2, something causes Library2 to load before Library1 (like creating an instance of B or using anything else in Library2), and then you create an instance of A, A's vtable would contain `_pure_virtual_called()` for all methods inherited from B and C. If Library1 was loaded before Library2 then this problem did not occur.

Fixed a bug that caused the ASLM gestalt selector to crash if ASLM had been installed and then the Shared Library Manager extension was remove from the extensions folder and the machine was rebooted.

Fixed a bug with `TArrayIterator::Next()` always returning `NULL` unless the iterator had a match object

In `NewObject()`, if the object's constructor throws an exception, then the object'‘s memory is freed and the object pointer is set to `NULL`. Previously the object pointer was still returned even though an error was also returned.

Fixed up Makefile.Examples.SC to fix a few problems and added a lot of Symantec related comments to ExampleClass.h.

Added some missing Symantec segments to the list of segments that BuildSharedLibrary and LinkSharedLibrary automatically merge when linking a shared library.

In LibraryManager.h, the `THINK_CPLUS` and `THINK_C` defines are checked along with `__SC__`.

In LibraryManager.h, the `MPWC` (now called `_CDECL`) in front of `extern "C"` routines was removed. It caused the Symantec C compiler to choke because it doesn't like `_cdecl` and it shouldn't be in front of `extern "C"` routines anyway.

In LibraryManagerClasses.h, moved the `TMemoryPool` inlines after the `TStandardPool` definition so SCpp doesn't complain about one of them doing a cast to a `TStandardPool`.

`TSCDyanmic::Trace()` was not declared as `_cdecl`. It is now.

`UnregisterLibraryFile()` will now first unload all libraries that are not in use before checking to see if there are any library's still in use in the library file.

Fixed a bug in the ASLM code that unloads a shared library. It was doing a `DisposeHandle` to dispose the shared library's heap and then doing a `DisposeHandle` and a `ReleaseResource` on memory within the heap that was just disposed of. This may have caused crashes with ASLM running with the Modern Memory Manager on PowerPCs. This bug was also fixed in ASLM 1.1.2.

Fixed a bug in the FixFMAP Installer action atom that sometimes caused the Installer to crash when installing into a System Folder that did not have a Finder.

`LoadSegmentByName()` and `LoadSegmentByNumber()` would cause the code segment to be loaded multiple times when called while the code segment was already loaded. This could cause the heap the code was loaded into to be used up so other code segments would not load.

`LoadSegmentByName()` and `LoadSegmentByNumber()` were not doing a cache flush after modifying the jump table and doing the 32-bit relocations. This would often result in strange crashes.

Segment number range checking was not being done by `LoadSegmentByNumber()` and `UnloadSegmentByNumber()`. This could cause a crash with `LoadSegmentByNumber()`. They will now both return `kASLMInvalidSegmentNumberErr`.

`UnloadSegmentByName()` and `UnloadSegmentByNumber()` were allowing you to try to unload segments in libraries that were built with the `noSegUnload` flag set (which is the default). This usually would eventually cause a crash. They will now return `kASLMNotSupportedErr`.

Fixed a bug in the LibraryBuilder macro processor that was causing it not to skip the last line of a multiline macro function. Regular multiline macros worked ok. Macro functions are ignored by ASLM, but since the last line was not being skipped, LibraryBuilder could get confused when reading the last line if it contained something that LibraryBuilder was looking for. In the case of the MacApp 3.1's `DeclareClassDesc` macro function in UClassDesc.h, the last line contained the code `"class" name` which led LibraryBuilder to believe that a class declaration had started.

Added macro support to all keywords that appear in an .exp file. See the "Corrections and Additions to ASLM 1.1.1 Documentation" section later in this document for more details on what macro processing features are supported.

Fixed a bug with the LibraryBuilder macro support of the `pascal` keyword before an exprorted function in the export list. If a macro substitution was used (like trying to define `_PDECL` to be either `pascal` or nothing at all and then using it in front the the function to export), LibraryBuilder would not expand the macro properly and give an error.

Made LibraryBuilder give a proper warning if the same class is exported twice rather than just saying that the class could not be found.

LibraryBuilder had a bug that could turn up if you intermixed #includes of class declrations with class exports in the .exp file. Once a class was exported, LibraryBuilder removed the class from an internal list of class declarations it had seen. This meant that if you later tried to #include a file that defined a subclass of the already exported class, LibraryBuilder would say that it had not seen the parent class's declaration yet. This bug has been fixed.

68k: Fixed a bug that caused ASLM to crash if a jump table was greater then 32K in size.

68k: Fixed a bug in LibraryBuilder that caused it to incorrectly report an error when a class that inherited from `SingleObject` had only one virtual function in its vtable. LibraryBuilder would complain that the class inherited from `SingleObject` but had an improper fromat for a `SingleObject` vtable.

Fixed a bug with LibraryBuilder not being able to deal with explicit exports of pascal methods.

Fixed both runtime and LibraryBuilder bugs with declaring C++ classes using the `struct` keyword and then exporting them. The runtime would crash if the struct had no virtual functions and LibraryBuilder would complain if there were any virtual functions.

Fixed LibraryBuilder so it always leaves at least 50k of Process Manager memory free.

Fixed `GetClientData()` so it can be called from the librar's InitProc and from the constructors of static objects.

In LibraryManagerUtilities.h, got rid of trailing ";" on `extern "C"` closing bracket and inline declarations. They were causing Metrowerks to give a warning.

Fixed an incompatibility between ASLM and newer versions of Stuffit SpaceSaver that caused system slow downs.

68k: Fixed a problem with Makefile.Examples.SC using the CFront version of LibraryManager.o instead of SCpp version and using an absolute path name to reference SCLibC.o.

68k: In LibraryManager.h, `MPWC` (now called `_CDECL`) was being defined as `_cdecl` when being included by Symantec C compilers. It should only be defined as `_cdecl` for Symantec C++ compilers and left empty otherwise.

68k: Fixed a problem with the math libraries in the MPW Libraries library file not working properly unless the client was compiled with the `-mc68881` option and was run on a machine with a math co-processor or the client was compiled without the `-mc68881` option and was run on a machine without a math co-processor.

68k: Fixed the ASLM Installer so it will install properly onto an A/UX machine.

Fixed the error message that LibraryBuilder gives if you have duplicate function set names.

Fixed `GetSharedResourceInfo()` so you don't need to `EnterSystemMode()` and `Preflight()` before calling it.

Fixed `FINALLY` macro. It used to not excute the code in the `FINALLY` section if an exception had been thrown.

Added `kTHashListGrowerID` to LibraryManagerClasses.h so you can call `LoadClass()` on it (which you need to do if you want to try growing a `THashList` at interrupt time).

Changed the `TException` declaration in LibraryManager.h to keep Metrowerks happy.

Fixed `THashList::Grow()`. It was causing the hash list size to double and was leaving interrupts disabled.

Fixed `TDoubleLong::RShift()` and `LShift()`. It was necessary to make them return a `TDoubleLong&` instead of a `TDoubleLong` and to actually modify the object rather than just returning a stack object with the result.

Fixed TraceMonitor crash that ocurred if ASLM was not available.

Fixed `CloseLibraryFile()` so it doesn't return an error if the file is preflighted unless the file is really about to be closed (it won't be closed if the usecount does not go to zero).

The "ASLM Symantec Support" folder no longer exists. The libraries that clients and shared libraries need to link with are now located in "ASLM Developer Tools:Libraries:SCLibraries:" for SCpp users and "ASLM Developer Tools:Libraries:THINKLibraries:" for THINK users. The SymbolConverter tool is now in the "ASLM Developer Tools:Tools:" folder.

The SCLibraryManager.o file has been renamed to LibraryManager.o.

Problems with some incorrect mangled names in the Symantec version of LibraryManager.o have been fixed. However, the SymbolConverter tool will still generate some incorrect names for certain mangled names. See the "Bugs and Warnings" section below for more details.

Declared `TClassID::operator==()` and the `TMatchObject` constructor as `_CDECL`. Otherwise the wrong calling conventions are used for SCpp clients.

Fixed a stack handling problems with pascal destructors (ones without `_cdecl` in SCpp or declared `pascal` in SCpp or CFront).

Declare `TCollection::Member()` and `Remove()` methods in a way that creates compatible vtables for both CFront and SCpp. CFront groups methods of the same name so they were not appearing in the vtable in the order that there were declared. However, SCpp always puts them in the order they are declared. They are now declared in the order that CFront has always been putting them anyway.

68k: Model far versions of `__vec_new` and `__vec_delete` are now included in the LibraryManager.o file. This solves the problem of shared libraries having to setup their A5 world before constructing an array of objects. The support exists for both CFront and SCpp.

Added a workaround to a resource manager bug that only occurs on the 660av and 840av when there are no fonts in the fonts folder. It was causing ASLM to not load.

LinkSharedLibrary will now make sure `{TempDir}` is always a full patch name, even if `{TempFolder}` and `{CPlusScratch}` are not set.

LinkSharedLibrary will now always first delete the temp files it is about to use in case they contain leftover stuff from another shared library build.

Fixed a bug in `CompareFileSpecs()` that caused it to not compare the last 2 bytes of the dirID field. This could cause 2 library files with the same name and on the same volume to compare the same even though they are in different directories. This would happen if the upper 2 bytes of the dirID for both files were the same. This bug had the side affect of causing about one out of every 40 library files installed by the Installer that were also replacing older file to not be registered with ASLM after rebooting for the first time. This had to do with the fact that the Installer Cleanup code would delete the old library file once the Finder started up and ASLM would think it was deleting the library file in the Extensions folder if both the old and the new file had the same has hash value (which they would about 1 in 40 times). After rebooting a second time everything would be ok.

LibraryManager.h no longer `#defines OLDROUTINENAMES`. It never should have.

All of the examples have been changed to no longer rely on `OLDROUTINENAME` being defined.

BuildSharedLibrary will now make sure it always uses a full path name for the batch file in case another file with the same name already exists in the `{Commands}` path, possibly because of a previously failed BuildSharedLibrary.

Fixed a problem with LibraryBuilder giving a bogus warning if it encountered a `#if <AnyNumericValue>` soon after a section that started with `#ifdef __cplusplus`. It would incorrectly warn you that it found a `#if __cplusplus` when it found the `#if <AnyNumericValue>`.

The newer ASM assembler puts the mod date of the assembled file into the object file which messes up LibraryBuilder when it tries to compare a newly created client object file with an older one. The newer one would always compare as being different so it would always replace the older one. The mod date of the <LibraryName>.stubs.a file is now always set to the same time before it is assembled so the same time stamp will appear in every client object file.

In LinkSharedLibrary, the `%Complex` segment was not being merged properly. It is now.

The ASLM Installer will now install the EtherTalk Phase 2 extension if it is not already present. This is to fix an incompatibility between ASLM and the version of EtherTalk that is in ROM on some cpu's.

Fixed a problem with `LeaveCodeResource()` accessing globals even after `CleanupLibraryManager()` has been called. This usually is only a problem if you are using the ZAP DCMD.

`TFunctionSetInfo` is now typedef'd to `TClassInfo` rather than just doing a forward class declaration that never gets defined.

Fixed the mask for the Shared Library Manager icon. It had a slight glitch in the bottom row of pixels.

Fixed a problem with the MPW Libraries support that turned up if you tried using `fopen()` and some other i/o related routines.

Fixed a problem with the MPW Libraries support that caused some of the libraries in the MPW Libraries file to never unload once they have been used. You are now required to call `CleanupMPWLibraries()` (defined in MPWSharedLibs.h) before calling `CleanupLibraryManager()` if you want to make sure that all the libraries unload.

Fixed a problem with the LibraryBuilder tool generating 4 characters of garbage in front of temporary file names if `{TempFolder}` and `{CPlusScratch}` are not set and the `-y` option was not used.

The Inspector, TestTools, and ExampleLibrary examples can now all build with either MrC, CFront, or SCpp. There is a separate makefile for each compiler type.

Added `DummyVirtualFunction()` implementations to the Symantec version of LibraryManager.o for all base classes (`TDynamic`, `TStdDynamic`, `TSCDynamic`, `TSimpleDynamic`, `TStdSimpleDynamic`, `TSCSimpleDynamic`, and `MDynamic`). These used to be omitted, causing link errors unless the user provided their implementations.

Made LinkSharedLibrary exit with an error if the Link or PPCLink command fails. It used not abort, which caused builds to continue when they shouldn't have.

`GetFunctionPointer()` and `GetIndexedFunctionPointer()` now return a `CDECLProcPtr` instead of a `ProcPtr`. The `CDECLProcPtr` typedef is the same as `ProcPtr` when using C, CFront, and SC. When using SCpp or MrC, it contains the `_cdecl` keyword, mainly to force C calling conventions when using SCpp.

Fixed a bug that showed up if an application did an `_ExitToShell` without first calling `CleanupLibraryManager()` and the ZAP DCMD was being used. ASLM's `_ExitToShell` patch calls `CleanupLibraryManager()` which then disposes the memory occupied by the patch. Before returning to the patch, the ZAP DCMD would have zapped it's code already and cause a crash. `CleanupLibraryManager()` will no longer free the memory occupied by the patch if it is being called by the patch. Instead it relies on the fact that the application's heap is about to be freed anyway so there is no need to free the patch's memory.

LibraryBuilder will now define the `applec` macro.

LibraryManager.h no longer defines the `powerpc` macro and does not use the macro anywhere in the header files or sample code. Instead, the new `GENERATINGPOWERPC` and `GENERATING68K` macros are used.

Function sets and classes may now have both `exports=` and `dontexport=` clauses as long as they also have `privateexports=*`. This allows you to export some functions/methods publically, explicity not export some, and have the rest exported privately.

LibraryBuilder will now produce an error message if you export a class method from a function set and the class is also exported and the method is not listed in the `dontexport=` clause.

Fixed some logfile support problems that caused either extra or missing warnings and error messages.

Fixed some logfile problems that turned up when using SCpp and MrC, that caused incorrect logfiles to be generated for any class containing pure virtual methods.

Fixed a problem with the logfile support that caused it to give an error message if you stopped overriding an inherited method.

Fixed a bug that caused `DummyVirtualFunction()` to always be exported from classes that use the `ASLM_SCDECLARATION` macro. This caused an extra entry at the beginning of the class's export table which would make the class no longer be compatible with older versions if you switch from building it with SCpp to CFront or vice-versa.

The 68k version of ASLM is now built with SCpp instead of CFront. It is still compatible with clients and shared librireas built using CFront, including those built with earlier versions of ASLM.

Since the built-in ASLM classes are now compiled using SCpp, SCpp clients and shared libraries can safely subclass them and also use them as stack objects and data members. Previously SCpp users could only create instances of the ASLM classes by using the new operator or NewObject() since the clases were compiled using CFront. This still holds true for any class provided by a 3rd party vendor that was compiled using CFront.

The libraries in the Libraries folder that used to support CFront have been moved to a subfolder called CFrontLibraries.

Made changes to the TMethodNotifier class so it can support being created by and both CFront and SCpp clients and can also call a CFront or SCpp method.

The SymobolConverter tool's -Symantec2MPW options was changed to -Symantec2CFront.

SCpp clients can now have static instances of ASLM classes.

Fixes a bug in the ShutDownActionAtom that could cause the Installer to crash older machines running System 6.0.5.

Fixed some problems with the `TScheduler` subclasses causing crashes with VM turned on. This normally only happend if the A5 world of the current application had been paged out when the TTimeScheduler fired and started processing operations.

Fixed a bug in `ResetFunctionSet()` that caused it not to work properly if you passed in a function set id that had a version number at the end of it.

New Features and Enhancements

- Added `#defines` to LibraryManager.h for the ASLM gestalt selectors (both PowerPC and 68K). If the ASLM gestalt selector returns 0 then ASLM did not load properly at boot time and is not available. Otherwise the version will be in the upper 2 bytes of the result and lower 2 bits of the result will contain information about the status of ASLM. `gestaltASLMVersionMask` is used to mask the upper 2 bytes of the gestalt result so that only the version bytes remain in the upper two bytes. `gestaltASLMPresentMask` is used to mask the lower bit to determine if ASLM is present or not. If this bit is set then ASLM was loaded at some point but may not be now. Calling `InitLibraryManager()` will cause ASLM to load if this bit is set and ASLM is not already loaded, except when calling `InitLibraryManager()` at boot time. In this case ASLM will set a flag so the next time the machine is restarted ASLM will stay loaded. After setting the flag the machine is rebooted. `gestaltASLMLoadedMask` is used to mask off the bit that determines if ASLM is currently loaded.

```
#define gestaltASLMPPC 'slmp'
#define gestaltASLM68K 'aslm'
```

```
#if powerpc
  #define gestaltASLM gestaltASLMPPC
#else
  #define gestaltASLM gestaltASLM68K
#endif

enum {
  gestaltASLMVersionMask   = 0xffff0000,
  gestaltASLMPresentMask   = 0x0001,
  gestaltASLMLoadedMask    = 0x0002
};
```

- Added `SetSystemAsClient()`. This call will make ASLM the current client. You should normally use it instead of calling `EnterSystemMode()` unless you really require that the system be in system mode. The only time you should need to be in system mode is if you have a shared library that needs to open a file that is not a library file and you want the file to remain open after the currently executing application quits. If you are calling `EnterSystemMode()` so you can preflight a library file without causing it to open a second time for the current client, then you should use `SetSystemAsClient()` instead. You can also use `SetSystemAsClient()` before calling `OpenLibraryFile()` when you need to make one of the `GetSharedXResource()` calls on a library file that is not currently opened by ASLM because none of the libraries in the library file are loaded.

  ```
  TLibraryManager* SetSystemAsClient();
  ```

- Added `GetSystemClient()`. It returns the client that represents ASLM itself.

  ```
  TLibraryManager* GetSystemClient();
  ```

- Added `GetLibraryClient(TLibrary*)`. It returns the client that represents the specified loaded library. You can then call `SetCurrentClient()` on the result if you want to make the library the current client. You can also use it to get the library's local pool by calling `GetObjectPool()` on the result or you can get the library's default pool by calling `GetDefaultPool()` on the result. If `GetLibraryClient()` returns `NULL` then the library is not loaded.

  ```
  TLibraryManager* GetLibraryClient();
  ```

- If the shift and command keys are held down while a shared library is being loaded, you will go into MacsBug the second time the shared library's entry point (`DynamicCodeEntry()`) is called (it's called 4 times for 68k and 5 for PowerPC). The debugger string will give the library id of the library. On PowerPC this will force the PowerPC debugger to "see" the code fragment (the debugger nub will also do this if you hold down the control key) and will allow you to setup break points in your shared library as soon as it is loaded and before any calls actually enter the shared library.

- If the shift and option keys are held down while a shared library is being loaded, you will go into MacsBug just before the library's code is loaded into memory with a message giving the library's id and the heap that the code will be loaded into.

- Sped up the loading of dependent shared libraries. When a client first uses a class or function set, a lot of work is done to make sure the library and its dependents are loaded if the `loaddeps` library flag is set. This work was being done even if the library had already been loaded. None of this extra work is done anymore once it has been done once.

- The "Tools" menu was added to the Inspector to provide some of the functionality of the TestTool MPW tool.

- Added the `-preservetemps` option to BuildSharedLibrary. This will preserve a copy of all temporary files created by BuildSharedLibrary. These files are mainly of use to the ASLM engineers to help debug shared libraries. They will be stored in the folder specified by the `-obj` parameter. Assuming you used `-obj MyLib`, the following files would be created:

  ```
  MyLib.bat
  MyLib.cl.o
  MyLib.deps
  MyLib.ia.o
  MyLib.init.a
  MyLib.init.c
  MyLib.init.o
  MyLib.lib.o
  MyLib.libr.r
  MyLib.stubs.a
  ```

- The Inspector now remembers the size and location of all windows. The means you don't need to relocate and resize windows so you can see them better each time you relaunch the Inspector. Also, Refresh menu items were added so you can refresh the front window or all windows. This allows you to see changes in usecounts without having to quit and relaunch the Inspector.

- Made `GetFunctionPointer()` and `GetIndexedFunctionPointer()` about 20 times faster when the library is already loaded.

- Added the `-fixp` option to BuildSharedLibrary. It is a work around to a bug with CFront. CFront doesn't uppercase methods that are declared pascal. `-fixp` will convert all pascal method names to all uppercase.

- Applications can now have static objects that are instances of ASLM classes. This wasn’t allowed before because static objects are constructed before the application’s main entry point is called and you need to call `InitLibraryManager()` before constructing any ASLM classes. Changes were made to the application startup code to allow you to call `InitLibraryManager()` before static objects are constructed. You need to do the following to get ASLM static objects to work in applications.

1. Implement a routine called `CallInitLibraryManager()` and have it call the `InitLibraryManager()` routine and return the result. `CallInitLibraryManager()` is declared in LibraryManagerUtilities.h. An empty implementation is provided in LibraryManager.o and LibraryManagerPPC.o so you need to be sure to link your version in first. `CallInitLibraryManager()` will be called by the application startup code before static objects are constructed.

2. Remove all calls to `CleanupLibraryManager()`. If you call `CleanupLibraryManager()` before the application exits, the application's exit code will crash when it tries to call the destructors for ASLM static objects. `CleanupLibraryManager()` will automatcially be setup as a routine to be called by the application's exit code when the application quits. In this case `CleanupLibraryManager()` is called after the static destructors are called.

3. For 68k applications, you need to link LibraryManager.o before Runtime.o. LibraryManager.o contains a modified version of `__Cplusinit()` which will call `CallInitLibraryManager()` before calling the constructors of static objects. If `CallInitLibraryManager()` returns an error then static objects will not be constructed.

   For PowerPC applications, you need to make `__ASLMcplusstart()` your application's entrypoint rather than `__cplusstart()`. `__ASLMcplusstart()` will call the `CallInitLibraryManager()` routine and then call `__cplusstart()` if `CallInitLibraryManager()` returns `kNoError`. If `CallInitLibraryManager()` returns an error then `__ASLMcplusstart()` will call `__start()` and static objects will not be constructed. The PPCLink documentation contains more information on `__cplusstart()` and `__start()`.

- The `-near` BuildSharedLibrary option has been replaced with `-nearClientFile`. Likewise, `-far` was replaced by `-farClientFile`, `-privateNear` was replaced by `-privateNearClientFile`, and `-privateFar` was replaced by `-privateFarClientFile`. The old options are still supported for backwards compatibility.

- The `-clientFile` and `-privateClientFile` BuildSharedLibrary options were added. Their main purpose is for specifying client object files when building PowerPC shared libraries (using `-farClientFile` didn't really make sense). However, they have the same meaning as `-farClientFile` and `-privateFarClientFile` and can also be used for 68k builds.

- Added support for `emulated` and `!emulated` flags in the Library declaration. If `emulated` is specified then the library will only be registered in cases where it will be run under emulation on a PowerPC. `!emulated` has the opposite affect of causing the library to only be registered on 68k machines. This flag is only supported for 68K shared libraries.

- Changed the 68K `TBitMap` class and also the `SetBit`, `TestBit`, and `ClearBit` routines so they treat bits from left to right within each byte instead of from right to left. In other words, 0 is now the left most bit within a byte. This way the bitmap layout is the same for both 68k and PPC ASLM.

- Made the 68k inline versions of the `AtomicXXXBoolean()` funtions return `1` for `true` instead of `0xFF`.

- `#if` processing is now allowed in the .exp file within `Class`, `FunctionSet`, and `Library` declarations. `#if` processing was previously only only allowed outside these declarations.

- In LibraryManager.h, the `Volatile` macro was renamed to `VOLATILE` and `VOLATILE` was renamed to `Volatile` so now `VOLATILE` is the same as MacApp's `VOLATILE`. Wherever you were previously using `Volatile` you should now use `VOLATILE`.

- In LibraryManger.h, `ClassID()` was renamed to `CastToClassID()`. This was done so there is no conflict with MacApp's `ClassID` definition. ASLM will `#define ClassID` to be the same as `CastToClassID`, but only if it is not already `#defined.` This means that if you are not using MacApp you can still use `ClassID()` and if you are using MacApp you cannot use `ClassID()` and should `#include` the MacApp headers before LibraryManager.h.

- Added `GetObjectsVersion()` and `GetObjectsMinVersion()` which will return the maximum and minimum version supported by the class that the object is an instance of. There are also method versions of these routines for all the `TDynamic` classes. As with the other `GetObjectsXXX()` routines, the object must be an instance of a class exported from a shared library and must have its vtable first.

- It is no longer necessary for a shared library to link with it's own client object file (.cl.o file), even if the library exports C++ classes. BuildSharedLibrary will now instead place the functions that the library needs from the client object file into another file that is already linked with the shared library automatically (the .init.a file).

- `FSInfoGetParentID()` was renamed to `FSInfoGetInterfaceID()` since this name makes more sense. `FSInfoGetParentID()` is still supported with a `#define`.

- Added the following functions which can be used to lookup a library’s `TLibrary*` by using either the library’s library ID, a class ID of a class in the library, or a function set ID of a function set in the library.

```
#ifdef __cplusplus
  TLibrary*  LookupLibrary(const TLibraryID&);
  TLibrary*  LookupLibraryWithClassID(const TClassID&);
  TLibrary*  LookupLibraryWithFunctionSetID(
                    const TFunctionSetID&);
#else
  TLibrary*  LookupLibrary(const TLibraryID);
  TLibrary*  LookupLibraryWithClassID(const TClassID);
```

```
    TLibrary*   LookupLibraryWithFunctionSetID(
                        const TFunctionSetID);
  #endif
```

- Added `TDoubleLong::GetWide` so you can extract both 4-byte longs out of the double long.
- `NewObject()` now takes a `TMemoryPool*` parameter rather than a `TStandardPool*`. This will not affect existing code at compile time or runtime since `TStandardPool` is a subclass of `TMemoryPool`.
- BuildSharedLibrary will now abort if LibraryBuilder reports an error.
- Added `-useLibraryID` and `-symdir` options to BuildSharedLibrary. `-symdir` is used to specify which directory to put the symfile in. It can be used with `-symfile` if the entire path name is not specified with `-symfile`. `-useLibraryID` is used to tell BuildSharedLibrary to use the shared library's library id as the symfile name. You can use `-symdir` to specify which directory the symfile should go in. Commas and colons will be converted to underscores and .SYM or .xSYM will be concatentated to the file name.
- The Symatnec SCpp compiler is now officially supported and the support is no longer considered experimental. However, there are still a number of bugs. Please read the "Symantec Products and the ASLM" document for more details.
- Added the `-cfrontCompatible` option to BuildSharedLibrary. It allows you to build SCpp shared libraries that can be used by CFront clients. It does this be adding constructor stubs to the shared library that will allocate memory for the object if no memory is being passed in (which is the case when a CFront client creates an instance of a class in an SCpp library and uses the default `new` operator). The penalty for using this option is 3 extra instructions being executed, even if memory is passed into the constructor.
- Under System 7 and later, ASLM will now no longer stay loaded at boot time unless there is a preloaded shared library that stays loaded. This is a feature that used to be in ASLM 1.1 but was removed before it was released. When an application calls InitLibraryManager(), ASLM will be loaded and will stay loaded until the next reboot. If an Extension tries to call InitLibraryManager(), ASLM will set a flag in it's preference file that will cause ASLM to stay loaded the next time the machine is restarted. After setting the flag the machine is immediately restarted so ASLM can be made available for the Extension. Once the "stay loaded" flag is set in the ASLM Preferences file, ASLM will always stay loaded so there will be no more machine restarts.
- Both the 68k and PowerPC versions of ASLM share the same variable for keeping track of the current nesting level of EnterInterrupt() calls. This allows an interrupt serviced by one ISA to call EnterInterrupt() and then make a mixed mode code to the other ISA at which time ASLM could be used without having to call EnterInterrupt first. The variable used for determing if InInterruptScheduler() should return true is also shared.

- BuildSharedLibrary now assumes that you are using the Symantec tools when building a 68k shared library and MrC when building a PowerPC shared library. If you are using CFront then you will need to pass the -cfront option to BuildSharedLibrary.
- BuildSharedLibrary and LinkSharedLibrary now assume that you are using PPCLink 1.2 or later and you no longer need to do anything special to get PPCLink 1.2 to work with ASLM. If you are using an older version of PPCLink then you will need to pass the -oldPPCLink option to BuildSharedLibrary and LinkSharedLibrary.
- Added the UnloadUnusedLibaries() routine. It forces any libraries the are currently scheduled to be unloaded (because they are no longer being used) to be unloaded immediately rather then letting ASLM wait up to 1 second before unloading them. This routine is rarely needed, but may be necessary if you have complex library dependencies and are having trouble getting your libraries to unload.
- Resources in library files that have the preload attribute set are no longer preloaded when opening up the library file under Single Finder. Single Finder provides very little extra System heap space and the System heap will not grow. Allowing preloaded resource to load was causing most, if not all, of the System heap memory to be used up.
- Added `TScheduler::IsSchedulerClientValid()`, `GetSchedulerClient()`, and `SetSchedulerClient()`. With the 68k version of ASLM, these fields actually act on the GlobalWorld saved with the `TScheduler` since their is no field for the saved client as there is for the PowrePC verson of ASLM. This means that `GetSchedulerClient()` will return the client that owns the GlobalWorld saved with the scheduler and `SetSchedulerClient()` will set the saved GlobalWorld to the global world belonging to the client passed to it.
- Added `TOperation::GetSavedClient()`, and `SetSavedClient()`. As with the new `TScheduler` routines by the same name, when using the 68k version of ASLM, these methods actually act on the GlobalWorld saved with the `TOperation` since only the PowerPC version has the saved client field.
- Added `GetClientFromWorld()`. It returns the `TLibraryManager*` the owns the GlobalWorld passed as a parameter. It is only available to 68k clients and shared libraries.

Corrections and Additions to the ASLM 1.1.2 Documentation

Using the "volatile" macros

Normally the `volatile` keyword is used for marking a variablle as being volatile. However, this keyword does not work with MPW C++. For this reason ASLM created the `VOLATILE` macro which simply creates a reference to the address of the variable and effectively makes the variable volatile when used with MPW C++. However, this trick may not work with all compilers and is known not to work with Symantec C++. For this reason, if you are compiling code with both MPW C++ and another compiler, you should use both the `VOLATILE` macro and the `Volatile` macro together to guarantee that the variable is treated as being volatile. For example:

```
  Volatile int  myint;
  VOLATILE(myint);
```

The `Volatile` macro will be defined as `volatile` for any compiler that supports the keyword. The `VOLATILE` macro will make sure that the variable is treated as volatile when compiled with MPW C++

Possible Memory Leaks When using overload new operators or when using MrCpp or SCpp.

If an overloaded new operator is being used or if MrCpp or SCpp is being used, the following code will produce a memory leak if an exception is thrown during the construction of `TDummyClass`.

```
  TRY
     TDummyClass foo = new TDummyClass;
  ENDTRY
```

The leak will occur because the memory for `TDummyClass`will be allocated by the above code before the constructor is called, rather then letting the constructor allocate the memory. If an exception is thrown (most likely because the shared library for `TDummyClass` could not be loaded), then there will be no way to recover the memory.

This problem does not occur with CFront (unless overloaded new operators are used), because CFront allocates the object memory in the constructor, so if the constructor is never reached then no memory will be lost.

There are a number of ways to deal with this problem:

•Don't worry about it. If not much memory will be lost then you probably don't need to worry about it. This might be true if the attempted allocation of the object was the result of a user action (such as choosing a menu item). If the action fails, the user isn't apt to continuously repeat the process until eventually the memory leak is fatal.

•Avoid needing exception handling. You can make sure that the construction of `TDummyClass` will not throw an exception by first trying to load it's library by using the `LoadClass()` routine.

•Use `NewObject()` if possible. If you try to construct an object by using `NewObject()` and the library implementing the object can not be loaded, an error will be returned rather than throwing an exception.

•Use a stack object rather then creating the object with the `new` operator. Since the memory will be allocated on the stack, it will still automatically be cleaned up even if an exception is thrown while constructing the object. However, you will need to declare and use the stack object within the scope of the `TRY/ENDTRY` block.

•Allocate the object from a pool that you can safely dispose of after (or soon after) the exception is thrown. This way the lost memory gets freed up with the pool.

C++ Versioning Issues

Appendix D describes how versioning works with ASLM. There is some incorrect information with regards to C++. The documentation says that if you add a data member, you should change the major version number of the class (ie. from 1.1 to 2.0). This is still true. The documentation also goes on to say :

> "Auto (stack) objects, objects created with the nondefault `new` operator, imbedded objects, and objects that are instances of the subclass whose major version number changed, can only use the class with the same major version number as the version number for the class contained in the client object file that the client or shared library linked with."

This is also still true, but a better generalization would be that any client of the class who does not allow the constructor of the class to allocate its own memory will not be able to use the class if the major version number has changed. The reason is because if the size of the object has changed, only the constructor and any clients built with the newer class declaration will be able to allocate the correct amount of memory for the object.

As it turns out, the ONLY time the constructor is allowed to allocates its own memory is if it is be created as a result of a CFront client using the default `new` operator. If a non default `new` operator is used (like the ASLM `new` operator that takes a `TMemoryPool*` parameter) or if SCpp is used, then the memory is always allocated by the client. This means that SCpp clients will not be able to use a class if it's major version number has changed. The client will either have to be rebuilt with the newer class declaration or the old version of the class will also need to be available for it to use.

Using Arrays of Objects

When using an array of objects, the array must be allocated and deleted by the same shared library or client. CFront uses a cache pointed to by a global variable to maintain a table of allocated arrays so they can be looked up when an array is deleted so the size of the array can be retrieved. The use of these globals will not work between 2 different clients or shared libraries, so arrays of objects must be allocated and deleted within the same shared library or client.

Although it is safe to have different clients and libraries allocate and delete arrays of objects when using SCpp, it is not safe to mix SCpp and CFront code in this way since SCpp uses `__vec_new` and `__vec_delete` routines that are incompatible with CFront (`__vec_new` is used to allocate arrays and `__vec_delete` to delete them). For this reason it is advised that arrays of objects be allocated and deleted within the same library or client, even when using SCpp.

MrC allocates and deletes arrays of objects the same way as SCpp, so currently there is no problem with allocating and deleting the arrays in different clients or shared libraries. However, it is still advised that you do not do this in case in the future

ASLM provides support for more than one PowerPC compiler like it currently does for the CFront and SCpp compilers.

Returning C++ Objects

It is not safe to return an object from a method or function if the object is an instance of a shared class. Only return pointers to objects or references to objects . For example, `TMyClass*` and `TMyClass&` are ok as return types, but `TMyClass` is not.

Using ASLM Memory Pools from C

The `SLMNewOperator()` and `SLMDeleteOperator()` routines allow C users to allocate memory from a `TMemoryPool`. `SLMNewOperator()` is used for doing allocations and `SLMDeleteOperator()` is used for freeing memory. `SLMNewOperator()` will allocate memory out of the specified pool. If you pass in `NULL` for the pool then it will use default memory allocation. The default memory allocation is to allocate out of the the library's local pool if the library was built with `memory=local`, and out of the client's local pool if the library was built with `memory=client`. If you don't want to use default memory allocation then you can pass in a pool retrieve by calling `GetLocalPool()`, `GetClientPool()`, or `GetSystemPool()`.

```
void*  SLMNewOperator(size_t, TMemoryPool*);
void   SLMDeleteOperator(void*);
```

Declaring Virtual Overloaded Methods

When declaraing virtual overloaded methods for exported classes, be sure to group all of the overloaded methods together. CFront does not always place virtual overloaded methods in the vtable in the order that they are declared, but SCpp does. CFront always groups virtual methods together by method name. Thus, if you have 2 versions of method `A()` and two versions of method `B()`, and declared them in the following order:

```
  virtual  A();
  virtual  B();
  virtual  A();
  virtual  B();
```

They will appear in the vtable in the following order if compiled by CFront: `A()`, `A()`, `B()`, `B()`. However, the Symatnec SCpp compiler always places them in the vtable in the order they are declared, and SCpp clients of the class will expect them to be in order of declaration. This will cause incompatibility problems when an SCpp client calls the method of a CFront class or vice-versa. Grouping the overloaded methods together in the declaration will ensure that both CFront and SCpp will expect the vtable to look the same.

TDynamic::Dump() Correction

TDynamic::Dump() only sends the result of GetVerboseName() to the TraceMonitor's trace window if the debug version of ASLM is running.

RERAISE Example Correction

The example using RERAISE on page 7-27 should contain a semicolon after the RERAISE statement..

GetSharedResourceInfo() Clarification

When calling GetSharedResourceInfo() on a shared resource, you must use the same TLibraryFile* object as the one that was used to retrieve the resource with GetSharedResource(), GetSharedIndResource(), or GetSharedNamedResource().

TCollection::Remove() Correction

The documentation does not mention that the boolean result returned by TColleciton::Remove() indicates whether or not the object was successfully removed from the collection.

TCollection::AddUnique() Correction

The documentation does not mention that the boolean result returned by TColleciton::AddUnique() indicates whether or not the object was successfully added to the collection. kNoError is returned if there is no error. kASLMDuplicateFoundErr is returned if the object already exists in the collection. Other errors are also possible depending on the collection being used and the problem that occured (such as out of kASLMOutOfMemoryErr).

TArbitrator Correction

On page 8-5 in the section "Looking up objects and claiming tokens", it says: “If you want to be notified synchronously when the request completes, you can provide a TNotifier when you call PassiveRequest() or ActiveRequest()”. It should say “asynchronously” and not “synchronously”.

GetLibraryClientData() Correction

On page 5-16, the GetLibraryClientData() routine is described as returning the client data for the client making the call. It actually returns the client data for the current client. If the client making the call is not the current client and you want to get the client data for the client making the call, you will need to call SetSelfAsClient() first and then restore the current client by calling SetCurrentClient() after making the GetLibraryClientData() call.

`LoadSegmentByNumber()` Correction

On page 5-17, the `LoadSegmentByNumber()` routine is described as returning `kCouldNotLoadCode` if the segment number is invalid. It actually returns `kASLMInvalidSegmentNumberErr`.

Exporting Static Methods

You need to be careful when exporting C++ static methods. The best way to export them is through a function set. Then they can always be called safely. If you try exporting them by specifying them in the export list of a class you are exporting, then they can only be called if an instance of the class has already been created. For this reason the recommended way of exporting C++ static methods in through a function set.

Don’t Use Memberwise Initialization!

Memberwise initialization of an object can mess up ASLM’s usecounting scheme for shared libraries. Memberwise initialization is done when one object is copied into another and there is no copy constructor declared. Copy constructors are also sometimes referred to as `X::X(const X&)` constructors. Memberwise initialization is simply the copying of one objects contents into another and is generated automatically by the C++ compiler when necessary.

There are two common places where one object is copied into another using memberwise initialization. They both require that no copy constructor be declared. The first is when an assignment statement is used to assign one object to another. The second is when an object is passed as a pass by value parameter. When objects are passed by value, C++ will automatically create an instance of the class on the stack and use memberwise initialization to copy the object into the stack object and pass the stack object instead.

The problem is that when memberwise initialization is used to construct an object, no constuctor is ever called. However, when the object is deleted the destructor is called. ASLM relies on usecounting code it adds to the constuctors and destructors to maintain proper usecounts for shared libraries. If the constructor is not called for an object and the destructor is, this will cause the usecount for the shared library to be one less then it is suppose to be which may cause the shared library to unload prematurely.

Memberwise initialization is only done if there is no copy constructor declared. If there is then it will be invoked to copy objects rather then just inlining memberwise initialization code. In this case ASLM will not have a problem with usecounts.

ASLM declares `TDynamic::TDynamic(const TDyamic&)` as a private member function so this prevents any `TDynamic` subclass from being copied. Other classes in the `TDynamic` family do the same thing. A subclass of `TDynamic` may still declare its own copy constructor if it wants to enable pass by value and assigment for the class.

`TBitMap::SetFirstClearBit()` Clarification

The Documentation for `TBitMap::SetFirstClearBit(size_t, size_t)` does not clearly state that the first parameter is the first bit of the range of bits to check and the 2nd paramter is the number of bits to check, not the last bit to check.

`LoadSegmentByName()` and `UnloadSegmentByName()` Corrections

`LoadSegmentByName()` and `UnloadSegmentByName()` are documented as saying that the address passed to them must be that of a jump table, which means that the code taking the address of the routine must not be in the same code segment as the routine. It should also mention that taking the address of a C++ method will not work because C++ does not generate a jump table reference in this case. It generates an 8 byte data structure instead.

BuildSharedLibrary `-D` Option

BuildSharedLibrary accepts the `-D` option which is used to define a preprocessing symbol just as with C/C++ compilers.

Unloading Function Sets with Circular Dependencies

If you have two or more function sets that are circularly dependent on each other, its possible that they will not automatically unload even when there is no longer a client around that is using them. The following example demonstrates why the problem occurs and how to correct the problem so the shared libraries will always unload. The example assumes that you have Library A and Library B which each call each other and a client exists that calls Library A. If the following happens then the libraries will not unload automatically:

•When the client calls A, A gets loaded and the client places a usecount on A. This usecount will not be undone (and Library A unloaded) until the client either calls `CleanupLibraryManager()` or does a `ResetFunctionSet()` to remove the usecount.

•After the client calls into A, A calls into B. This will cause B to load and A will place a usecount on B. This usecount will not be undone until either A is unloaded or A calls `ResetFuntionSet()`.

•After A calls into B, B then calls back into A. A is already loaded, but this call will cause B to place a usecount on A. This usecount will not be undone until B either unloads or calls `ResetFunctionSet()`.

Now the libraries and client are in the following state:

•B will not unload until A either unloads or calls `ResetFunctionSet()`.

•A will not unload until B either unloads or calls `ResetFunctionSet()` AND the client calls `CleanupLibraryManager()` or `ResetFunctionSet()`.

After the client calls `CleanupLibraryManager()` or `ResetFunctionSet()` the libraries are in the following state:

•B will not unload until A either unloads or calls `ResetFunctionSet()`.

•A will not unload until B either unloads or calls `ResetFunctionSet()`.

What it all boils down to is that once the client calls `CleanupLibraryManager()` or `ResetFunctionSet()`, A and B will not unload until one of them calls `ResetFunctionSet()` on the other. It's up to the programmer to realize when this should be done. All `ResetFunctionSet()` does is remove any cached information a client may have about functions in a function set and removes the usecount that the client has on a function set. The penalty is that the next time a function is called, the function address will have to be looked up and cached again and the library will also have to be loaded if it is not already.

If your situation is as simple as the above example, then the easiest solution is to have the client make some sort of “cleanup” call into Library A just before it calls `CleanupLibraryManager()`. When Library A receives this "cleanup" call, it can just call `ResetFunctionSet()` (either passing in `NULL` or the function set in B that was used) and then return. When the client calls `CleanupLibraryManager()` then everything should unload. If you are using C++, you can also force a shared library to do a `ResetFunctionSet()` buy getting the `TLibraryManager*` for the library and then using it to call `ResetFunctionSet()`. There are a number of routines you can use to lookup a library's `TLibrary*` object and then you can call `GetLibraryClient(TLibrary*)` to get the library's `TLibraryManager*` object.

LibraryBuilder Preprocessor Capabilities

The preprocessor capabilities of the LibraryBuilder tool have never really been documented and some improvements have been made in ASLM 2.0 to make its abilities more complete and robust. Macro substitution is allowed almost everywhere now and #if/#else/#endif processing is also allowed everywhere that it is allowed in normal C/C++ code.

The following macro and #if processing examples will all work with LibraryBuilder:

```
#ifdef _powerc
  class Foo
#else
  class Foo : SingleObject
#endif
{
  …
}

#if USEPASCAL
  #define PDECL pascal
#else
  #define PDECL
#endif
```

```
#define MYHEAP application

FunctionSet MyFunctionSet
{
  …
  heap = MYHEAP;
  exports = PDECL FooFunction;
}
```

There are a few macro expansions that LibraryBuilder cannot handle. They include the following:

•Macro functions.

•Multiword macros.

•Macro substitution for the `Library`, `Class`, `class`, `struct`, and `typedef` keywords.

•Macro substitutions for numeric values such as for the `clientdata` and `heap` options.

The following examples are not allowed.

```
/* not allowed because it uses a macro function */
#define CLASSDECL(name, parent) \
  class name : public parent
CLASSDECL(Foo, FooParent)
{
  …
};

/* not allowed because it uses a multiword macro */
#define PARENT public MyBaseClass
class Foo : PARENT
{
  …
};
```

Macro functons and multiword macros are allowed in the .exp and .h files, but they will not be expanded. They are simply read in and ignored.

If you ever need C preprocessor functionality that the LibraryBuilder tool does not support, you can always run the .exp file through the C preprocessor and pass the output on to LibraryBuilder in place of the .exp file.

Bugs and Warnings

RUN THE INSTALLER!!! Don't try to drag install ASLM 2.0 over any previous release of ASLM. The resource that was installed into the System file by earlier versions may not be compatible with version 2.0. Also the 68k and PowerPC installers do not install the same resources into the System file so don't try running the installer for one and then drag installing the other.

You might have problems installing ASLM from a CD-ROM drive. If so, then make a floppy out of the "ASLM Installer.image" file and use this as the installer disk. You can also create a floppy by copying the CONTENTS of the "ASLM Installer" folder to floppy and naming it "ASLM Installer".

Installing any version of System 7 when ASLM is already installed will cause ASLM to no longer load. You will need to re-install ASLM in this case.

ASLM will not load under System 7.0.1 with tuneup 1.1.1 installed and AppleTalk turned off. Users should either turn AppleTalk on or upgrade to System 7.1 or later to avoid this problem.

There is currently a bug if you have shared libraries with circular dependencies due to inheritance. The simple case of this is if A and C are in one shared library, B is in a second shared library, and C inherits from B which inherits from A. A more complex case would be if A and D are in one shared library and B and C are in a second shared library and A inherits from B and C inherits from A. You should try to avoid this situation, if possible, by simply moving some of the classes to other shared libraries so there is no longer a circular dependency like this. If you don’t want to do this then you need to take certain precautions. The easiest precaution is to simply do an explicit load on a class in the first library before allowing the second library to load. When you are finished with both libraries then you can undo the explicit load.

There's a bug in the Symantec version of `__vec_new()` that prevents clients from allocating an array of objects if the constructor is declared as _cdecl. The constructor will still end up being called with pascal calling conventions. If your client is only allocating arrays of objects that use `_cdecl` constructors, then you can change the implemention of `__vec_new()` and recompile it. You need to change the `ctor_t` typedef to include `_cdecl` in it. You will also need to change the `__vec_ctor()` routine in the same way. Both routines are located in vecnew.cpp. Make sure you compile them model far if they are to be used with a shared library. Note, if you make this change then you will not be able to allocate arrays of objects whose constructor is NOT `_cdecl`.

Multiple Inheritance does not currently work with MrC or SCpp. This still needs to be investigated and it is unclear if it will be fixed.

The SymbolConverter tool does not work for methods or functions with more than one parameter of the same type. You will need to rename these symobls using the Lib tool. This bug will most likely not be fixed.

SCpp classes cannot subclass CFront classes. This bug will not be fixed.

If SCpp code deletes an instance of a CFront compiled class that does not have a vtable, the memory will be deleted twice. This bug will not be fixed.

If CFront code deletes an instance of an SCpp compiled class that does not have a vtable, the memory will not be freed. This bug will not be fixed.

When using the PowerPC version of ASLM, you cannot create a TTimeScheduler or a TInterruptScheduler at interrupt time.