Apple Shared Library Manager Developer's Guide

Contents

Preface About This Guide

The *Apple Shared Library Manager Developer's Guide* documents version 1.1 of the *Apple Shared Library Manager* (ASLM). The ASLM is a set of software tools for developing and using shared libraries. The ASLM allows multiple programs, or clients, to share code, data, and resources stored in libraries. The ASLM supports applications and shared libraries written in all MPW-compatible languages, such as C, C++, Pascal, and assembly language.

To help developers create and use shared libraries, this guide contains general information on developing and using shared libraries, programming examples, and a reference of ASLM utility classes and functions.

Audience

The *Apple Shared Library Manager Developer's Guide* is intended for software developers who want to:

- use prewritten shared libraries
- design and create shared libraries

To use prewritten shared libraries, a software developer must understand the Macintosh Toolbox and Operating System, and know how to write programs in an MPW-compatible language such as C, C++, Pascal, and assembly language.

To design and create shared libraries, a software developer must also understand the operation of the Macintosh Memory Manager, and know how to write programs in MPW C++ (to write C++ classes).

Organization

The *Apple Shared Library Manager Developer’s Guide* is divided into four parts:

- Part I, “Overview and Installation,” which introduces shared library concepts and the ASLM, and describes how to install the ASLM. Part I consists of the following chapters:
  —Chapter 1, “Introduction to Shared Libraries”
  —Chapter 2, “Introducing the ASLM”
  —Chapter 3, “ASLM Installation”
- Part II, “Developing Clients and Shared Libraries,” which describes how to write and build clients and shared libraries. It also includes miscellaneous topics related to using the ASLM. Part II consists of the following chapters:
  —Chapter 4, “Writing and Building Clients”
  —Chapter 5, “Writing and Building Shared Libraries”
  —Chapter 6, “Using the ASLM”
- Part III, “Reference,” which describes all of the ASLM utility functions and classes. Part III consists of the following chapters:
  —Chapter 7, “ASLM Utilities”
  —Chapter 8, “ASLM Utility Class Categories”
  —Chapter 9, “Utility Classes and Member Functions”
- Part IV, “Appendixes,” which describes the following topics:
  —Appendix A, “Header Files”
  —Appendix B, “ASLM Utility Programs”
  —Appendix C, “Using the Example Programs”
  —Appendix D, “Versioning”

I Overview and Installation

1 Introduction to Shared Libraries

This chapter introduces the concept of shared libraries and explains the key features and functions of a shared library. If you are already familiar with shared libraries, you may want to skip ahead to Chapter 2, “Introducing the ASLM,” for specific information about the Apple Shared Library Manager (ASLM).

Shared libraries

A shared library is a library of functions or classes (for C++ programmers) that are compiled, linked, and stored separately from the clients that use them. By accessing the functions or classes that are stored in a shared library, a client can call functions that are not part of its executable code. Furthermore, functions or classes that are stored in a shared library can be called by different applications that are running at the same time.

Because shared libraries can contain shared code and are loaded and linked at run time, they save enormous amounts of RAM and disk space. Shared libraries eliminate the necessity for keeping multiple copies of code in memory when multiple applications use the same code.

Shared libraries help software developers design independent, modular, compact libraries that applications can share. It also helps software designers develop their products faster, and it makes the products easier to improve and maintain.

A *shared library file* is a binary file that can contain object code for functions, classes, methods (member functions), data, and resources. A shared library file can contain one or more shared libraries. When a shared library file is made available, developers can share, and dynamically link with, the code stored in the shared library.

A *client* is any application or library that creates objects or uses methods or functions that are implemented in shared libraries. Clients can include applications, system extensions, INITs, CDEVs, XFCNs and XCMDs, other kinds of stand-alone code resources, and even shared libraries themselves, because shared libraries typically use other shared libraries.

A client written in a non–object-oriented language, such as C or Pascal, can call routines that are stored in a shared library in the same way that it would call any other function. A C++ client can instantiate object from classes that are stored in the shared library in the same way that it would instantiate objects from any class.

Dynamic versus static linking

Although clients can use functions and classes stored in shared libraries as they would use functions or classes that are made available in ordinary libraries, shared libraries are compiled and linked differently from conventional applications and libraries. While applications and conventional libraries are *statically* linked, shared libraries are *dynamically* linked with the applications that use them.

Static linking takes place when the linker combines object-modules produced by a compiler into an executable program. Dynamic linking takes place at run time; that is, when an application is executed.

If an application needs to call a function or instantiate a class from a conventional library, the application must link with that library at the time the application is created. In this kind of linking—static linking—a copy of the library function that the application needs is placed in the application's executable file at link time. In this way, a separate copy of the library function is placed in the executable file of each application that uses the function. Figure 1-1 illustrates static linking.

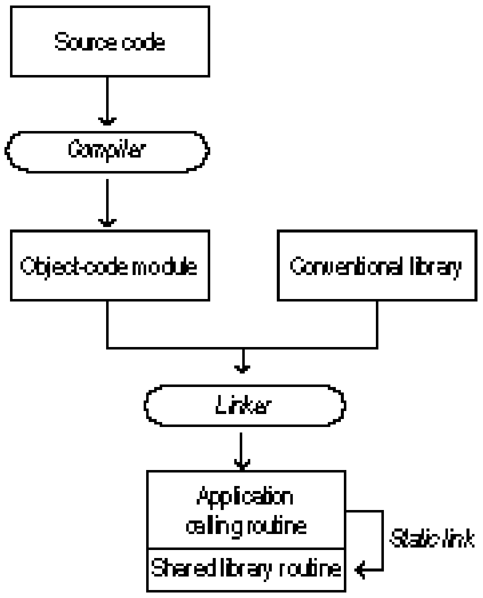


Figure 1-1 Static linking

In contrast, when an application needs to call a function or instantiate a class that is stored in a shared library, the shared library does not provide the application with its own copy of the code needed to execute the function or implement the class. Instead, at link time, a stub that tells the application where it can find the object code of the function or class in the shared library is placed in the application's executable file. At run time, the application uses this information to locate the function or class that is stored in the shared library. That process is called dynamic linking. Dynamic linking is illustrated in Figure 1-2.

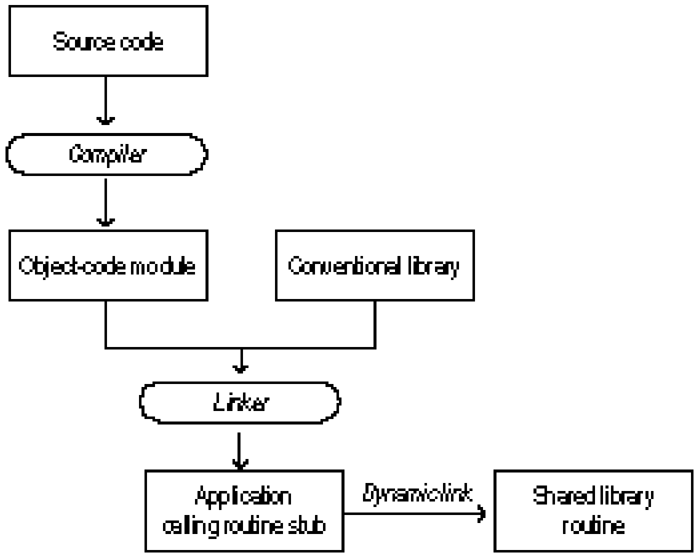


Figure 1-2 Dynamic linking

When applications and libraries are linked dynamically, multiple applications can use a single copy of an executable module simultaneously. As Figure 1-2 illustrates, the code in the shared library is stored separately in memory from the applications that use it. A shared library can (and usually does) contain multiple procedural-language functions or C++ classes, and any client can use any of these functions or classes as if they were part of the client's executable code. Therefore, shared libraries can save memory space.

Using shared libraries with object-oriented programs

For programs written in C++, the ASLM supports code reuse by dynamically linking and loading C++ class implementations and by supporting *dynamic inheritance*.

Dynamic linking and loading, together with dynamic inheritance, let multiple clients share the same implementation of a class. Dynamic linking lets a code module make direct calls to another code module's functions when the two modules are not implemented in the same code resource or object file. Dynamic loading also lets the shared library load the implementation of a class, on demand, at run time. Dynamic inheritance allows a subclass to be derived from a base class that is in another shared library.

These are some of the features of shared libraries used by C++ programs:

- *Dynamic class identification.* When you create an object that is implemented in a shared library, you do not have to hard-code its class ID. This means that a configuration or installation process can determine the specific class or set of classes to be used in a program.
- *Dynamic inheritance.* A class that is implemented in a shared library can inherit from a class that is not in the same shared library. This means that a developer can create a class that inherits from another developer's class.
- *Class verification.* A client can call a function to verify at run time that a given object is derived from a particular base class.

Updating code in shared libraries

A shared library is a "black box" to the applications that use it because shared libraries are kept separate from applications in memory and are accessed by applications using stubs. Clients can use the functions in a shared library without having access to the details of how the functions work. So, when code in a shared library is updated, the changes to the code have no effect on the applications that call the shared library; the changes are transparent to the library's clients, as long as the library's interface remains compatible.

Therefore, when you want to change a function or a class in a shared library—to improve its execution speed, for example, or to add more features—you can do so without recompiling or relinking the application. In fact, an update supplied by the developer of a library can split functions and classes that had previously been in one file into several files without any impact on the client, and no recompilation is necessary. The update can even occur while the client is running, provided the library is not already loaded. (If the older library is already in use by a client, the newer library is not used by current clients of the older library until the older library is unloaded. New clients will always use the newest version.) "Versioning" of libraries is used to determine which library is to be used when the same function or class is implemented in more than one library. (For more information about versioning, see Appendix D, "Versioning.")

Modifying code in shared libraries

Because shared libraries let you modify and enhance applications without having to rebuild them, you can change the behavior of a client by simply calling a different shared library. For example, if an application has access to several shared libraries that display the same data in different formats on the screen, the application can change from one screen display to another by simply using a different shared library.

In contrast, when an application is statically linked with a conventional library, the application must be relinked if there is any change in any functionality in the library.

In one respect, shared libraries do add a level of complexity to a program’s design. When an application uses functions stored in a shared library, the library must be stored in memory in a separate module. At run time, if an application cannot find the shared library in which the function is stored, it cannot execute the function.

2 Introducing the ASLM

This chapter introduces the features and benefits of the Apple Shared Library Manager (ASLM). Use this chapter to familiarize yourself with some important terms and features which will help you use the ASLM more productively.

What the ASLM can do for you

The ASLM is designed for software developers who want to develop libraries of routines (for non–object-oriented programs) or classes (for C++ programs) for use by multiple applications. The ASLM

- saves time in program development and maintenance
- simplifies the sharing of functions and C++ classes at run time, thus encouraging software developers to reuse code by providing libraries of functions and C++ classes that multiple clients can access simultaneously
- allows applications to share, reuse, and dynamically link code
- aids development of platform-independent applications
- can be used with any application, extension, or device driver—including an interrupt handler
- supports object-oriented (C++) and non–object-oriented (C, Pascal, and assembly) languages
- provides developmental and diagnostic tools
- offers expandability through the addition and use of new shared libraries

Suppose, for example, that your company wanted to design a text editor, a telecommunications program, and a fax modem driver. Text-editing routines designed for use by both the text editor and the telecommunications program could be placed in the same shared library. Communications routines for both the telecommunications program and the fax modem program could be placed in another shared library. Still another shared library could contain menu and window manipulation routines common to all three programs.

By giving your application access to these three shared libraries, you could save time in program development. Since routines implemented in the shared libraries could be shared by all three programs, customers running your applications could save disk space and memory. Your programs would also load faster, since they would share object code.

The ASLM significantly enhances the benefits offered by other shared library implementations, such as dynamic linked libraries (DLLs), which may be familiar to programmers who have worked in the Windows, OS/2, and UNIX® operating systems.

The ASLM is intended to help software manufacturers produce products that are better designed; easier to implement, test, and use; and after they are shipped, easier to enhance and maintain.

Some important terms and concepts

The following important terms and concepts are used throughout the document. They are explained in more detail in later chapters.

Client A client is any application, shared library, or stand-alone code resource that makes use of shared libraries. Shared libraries are always considered to be clients. Applications and stand-alone code resources become clients by making a special call to register themselves as a client with the ASLM.

Current client The current client is generally the currently executing application, but other clients (such as shared libraries and stand-alone code resources) have the ability to make themselves the current client also. The current client is generally used to determine on whose behalf something is done, such as allocating memory, opening a file, or setting up or making a callback.

Function sets Function sets are a set of C or Pascal functions that are implemented in a shared library and can be called from programs written in C, C++, Pascal, or assembly language. Any function that a shared library writer wishes to export must be placed in a function set.

Class ID The class ID is a C string that provides a unique identifier for a given class. For example, the class ID of a class called `TLinkedList` might be `ASLM$TLinkedList,1.1`. The class ID always starts with a four-character developer ID to ensure that it is unique and is followed by a dollar sign ($), text that helps describe the class (but does not have to be the same as the class name), and usually ends with a version number. Class IDs are used to determine which class a client should dynamically link with when using a class exported by a shared library.

Function set ID Like classes, function sets are also given an ID which is a C string that provides a unique identifier for a given function set. For example, the function set ID of a function that provided routines for maintaining a linked list my be called `ASLM$LinkedListFSet`. Like the class ID, the function set ID always starts with a four-character developer ID to ensure that it is unique and is followed by a dollar sign ($), text that helps describe the function set, and usually ends with a version number. Function set IDs are used to determine with which function set a client should dynamically link when calling a function exported by a shared library.

Client object file The client object file is a file that contains routines and information that is necessary to dynamically link a client with a shared library. Each shared library provides a client object file and most clients of a shared library must link with the shared library's client object file. The client object file contains things like function stubs for exported routines, including functions implemented in function sets and the methods of C++ classes. The client object file also contains the IDs of function sets and classes to be used.

Function stubs Function stubs, also called "glue" routines, are responsible for dynamically linking a client with a shared library. They are located in the client object file, and have the same name as the routine they are responsible for dynamically linking with. For example, if a C programmer calls a routine called `hello` which is located in a shared library, he will actually link with a stub routine called `hello`. The stub will take care of making the dynamic link with the shared library that implements `hello`.

Library ID The library ID is similar to the function set ID and class ID, except that it is used to represent a shared library. Library IDs are not used very often and are not contained in the client object file, but it is necessary for each shared library to have a unique library ID.

Library files Library files are files that contain one or more shared libraries. Each shared library will have its own set of code resources and other resources such as a `'libr'` resource that provides information about the shared library. It is important to realize that a shared library is not represented by a file, but by a set of code resources located in the file and the `'libr'` resource that describes the shared library. A library file may contain more than one shared library.

Model near and model far Model near and model far are terms used to describe how executable code (such as an application, stand alone code resource, or shared library) is built. In brief, model near executables use 16-bit A5 relative references to access global variables and to make intersegment subroutine calls using the jump table. This means that A5 always needs to be set up properly before accessing globals and making an intersegment call. This is the way all executables used to be built until model far was introduced. Model far executables have all global and jump table references resolved to absolute 32-bit addresses when the code segment is loaded, so it is usually not necessary to have A5 set up, although model far executables still require an A5 world (global world).

It is important to realize that all shared libraries are built using model far and shared library clients can be model far or model near, although MPW requires that all stand-alone code resources be model near. For more information on model near and model far, refer to the latest MPW documentation and release notes.

Features of the ASLM

This section introduces many of the important features and capabilities of the ASLM.

Creating C++ objects by using class ID's The ASLM supports creating a C++ object by specifying the class id of the class that the object is an instance of. This allows the programmer to decide at runtime which class to instantiate .

Calling functions by name or index The ASLM supports calling a function by supplying the function set ID plus either the function name or the index of the function in the function set. This is useful for code ported from other DLL solutions, and for applications such as spread-sheet macros and scripting-language extensions.

Finding all classes with a common base class The ASLM allows you to find all classes with a common base class. This allows you to decide at runtime which classes are available to support your needs.

Finding all function sets with a common interface The ASLM allows you to find all function sets with a common interface. This allows you to decide at runtime which function sets are available to support your needs. When a function set is built, it can specify an interface id. Function sets with a common interface can share the same interface id. This allows you to locate all the function sets with the same interface id so you can then choose which function set you want to use.

Dynamic installation of libraries Libraries can be made available after boot time by dragging the library's library file into any registered library file folder, including the Extensions folder.

Access to object meta information The ASLM allows you to access information about a C++ object such as the class id of the parent(s) of the object and what shared library the object is implemented in.

Multiple inheritance The ASLM fully supports multiple inheritance of C++ classes.

Client death notification The ASLM provides a notification facility that you can use to determine when a client goes away. A client goes away when a client application quits or a shared library is unloaded. To keep track of when clients go away, you can register what is known as a *death watch* notifier or *death watcher*.

Exception handling The ASLM provides exception-handling macros that are used to catch exceptions that may be raised. The only time the ASLM will raise an exception is if it fails to load a shared library or fails to load a shared library's code segment after the shared library has already been loaded. The ASLM's exception-handling macros match the DCE standard and can be used from C.

Explicit segment unloading support The ASLM supports the explicit loading and unloading of library segments by the library or library client.

Languages supported by the ASLM Shared libraries can contain function sets for C, Pascal, and assembly language programs, as well as implementations of C++ classes.

C++ programs can create objects and call methods that are implemented in shared libraries. Programs written in non–object-oriented languages can also call methods implemented in shared libraries, but only if the developer of the shared library provides a special procedural interface for the class.

Library loading and unloading Explicit loading and unloading of libraries is supported to ensure that a shared library is available.

Pascal header files The ASLM provides LibraryManager.p and LibraryManagerUtilities.p interface files that list most of the routines that are currently available to C programmers.

Per client data Per client data is supported by a simple mechanism that allows a library to have a separate data structure for each client. A library simply calls a routine to get the data structure for the current client and a client can call a routine to get its data structure for a specified library.

Preloading all dependent libraries To facilitate the easy preloading of libraries on which a client depends, the ASLM provides an MPW tool which generates a resource containing information about all the dependent libraries, and provides a routine that will load all libraries described in the resource.

Registered library files The ASLM supports the registration of any file as a shared library file.

Registered library file folders The registration of folders in which library files can be located is supported. These folders support dynamic installation of library files in the same way as the Extensions folder.

Snap-linking To speed up processing and to provide an efficient calling mechanism, shared libraries are “snap-linked.” Snap-linking is an address-caching technique in which binding overhead usually occurs only once. After binding occurs, the target address is cached, so the link can be “snapped” in the client.

This calling mechanism is very efficient and makes programs load and run faster. It is particularly well suited for the kind of time-critical use that is required by high-performance networking protocols or timing-dependent device drivers.

System support The ASLM supports systems 6.0.5 and higher. There are some limitation when using ASLM under System 6. See “Using the ASLM Under System 6 and System 7” in Chapter 6 for more information.

Utility classes provided with the ASLM The ASLM comes with a collection of utility classes that you can use in your own applications and shared libraries. These utility classes can be divided into the following categories:

- Memory-management classes are a set of memory allocation classes called memory pools, which are special pools of memory that shared libraries and clients can use in place of memory normally allocated by the Macintosh Memory Manager. The particular advantage of these classes is their speed and the fact that they are interrupt-safe.
- Collection classes keep track of objects in different types of collections, such as arrays, hash lists, and linked lists.
- Object arbitration classes let multiple clients share named objects.
- Process management classes let you schedule tasks to run during System Task time, at interrupt time, or at predetermined intervals.
- Library file and resource management classes allow clients and libraries to access resources in a shared library’s file.
- Miscellaneous classes include timing classes and other kinds of classes, such as random number classes (used for generating random numbers in a variety of different ways), that are used for essential operations by the ASLM and can also be used by clients.

Versioning The ASLM enables the specification of the version numbers of the shared library, function sets, and classes implemented in the shared library or used by the client. The ASLM uses the function set or class with the newest version number that is also compatible with the version specified in the client object file with which the client linked.

3 ASLM Installation

This chapter provides installation instructions for the ASLM and associated development tools, and describes the contents of the ASLM disks.

The ASLM developer’s kit is distributed on the following four disks:

- The *ASLM Installer* disk that contains the Shared Library Manager extension file that oversees all the functions of the ASLM. It also contains the installer script that installs the ASLM onto your system.
- The *ASLM Developer Tools* disk that contains the tools, scripts, and header files that you need to write, compile, and link your own shared libraries and clients under MPW.
- The *ASLM Debugging Tools* disk that contains debugging applications, such as the Inspector application and the TraceMonitor application.
- The *ASLM Examples* disk that contains example programs that can help you learn how to develop and build shared libraries.

Installing the ASLM

To install the ASLM, run the Installer application located on the *ASLM Installer* disk. The Installer places the Shared Library Manager extension file in your System 7 Extensions folder (or System 6 System Folder). The Installer also installs resources in the System file and performs other essential house keeping operations. The Installer must make these modifications to your System file before you can use the ASLM.

To install the ASLM, follow these steps:

1 **Open the ASLM Installer disk and double-click the Installer icon.**

2 **Click Install.**

The Installer places the Shared Library Manager extension file in your System 7 Extensions folder (or System 6 System Folder) and performs other essential installation operations.

3 **Click Restart when the Installer is finished.**

You can now execute any client; that is, any program that makes use of shared libraries.

The Installer does not install the tools needed to develop and debug shared libraries. You can install those tools as described in the following sections.

Installing the developer tools

To develop shared libraries, you must have MPW 3.2 or later installed on your hard disk. If you are going to develop any shared classes, you must also install MPW C++ 3.2 or 3.3. (Shared classes are classes that the ASLM knows about because they are in a shared library.) Once MPW is installed, you can copy the tools that are needed to develop shared libraries from the *ASLM Developer Tools* disk into your MPW folder.

The *ASLM Developer Tools* disk contains a Read Me! file and three folders with tools and utilities that you can use to develop your own shared libraries. The Read Me! file contains information regarding the contents of the disk. Some of the files in the three folders are essential for developing shared libraries; others are utilities that you may find useful.

To copy the tools onto your hard disk, follow these steps:

1 **Open the** ASLM Developer Tools **disk.**

2 **Open the Tools folder.**

3 **Drag the MPW scripts—**`BuildSharedLibrary` **and** `LinkSharedLibrary`**—into the MPW Scripts folder.**

4 **Drag the MPW tools—**`LibraryBuilder` **and** `CreateLibraryLoadRsrc`**—into the MPW Tools folder.**

5 **Open the Libraries folder.**

6 **Drag the five MPW libraries into the Libraries folder in your MPW Libraries folder.**

The five libraries are LibraryManager.o, LibraryManager.n.o, LibraryManager.debug.o, LibraryManager.debug.n.o, and TestTool.o. If you are a THINK C/C++ user, drag the THINK Libraries onto your hard drive.

7 **Open the Interfaces folder.**

8 **Drag the files from the CIncludes folder into the MPW CIncludes folder.**

9 **Drag the files from the PInterfaces folder into the MPW PInterfaces folder.**

10 **Drag the files from the RIncludes folder into the MPW RIncludes folder.**

You now have all the tools you need to develop your own shared libraries and shared library clients.

Note: You do not have to install the ASLM tools, scripts, interfaces, and MPW libraries into your MPW folder. However, you will have to set up MPW and your makefiles to locate the ASLM files. This can include adding the directory containing the tools and scripts to the MPW {Commands} shell variable and specifying the location of the interface files by using the `-i` option when compiling your code. Also, the `BuildSharedLibrary` and `LinkSharedLibrary` scripts automatically look in the {SLMTools} folder for the `LibraryBuilder` and `CreateLibraryLoadResource` tools.

Installing the debugging tools

The *ASLM Debugging Tools* disk contains a Read Me! file, three folders, the Shared Library Manager Debug extension, the ASLM Debugger Prefs ResEdit document, and the TraceMonitor application. The Read Me! file contains information regarding the contents of the disk. The files on this disk are useful in debugging ASLM clients and shared libraries.

1 **Open the** ASLM Debugging Tools **disk.**

2 **Drag the TraceMonitor application onto your hard disk. You will need to run this application while debugging.**

3 **Copy the resources from the ASLM Debugger Prefs file into your MacsBug Debugger Prefs file, which should be located in your System Folder.**

You can use ResEdit or the MPW Rez tool to copy the resources. Perform this step only if you use MacsBug.

4 **Open the Inspector folder.**

5 **Drag the shared libraries—InspectorLibrary and WindowStackerLibrary—into your Extensions folder.**

6 **Drag the Inspector application to a place on your hard disk. You will need to run this application while debugging.**

7 **Open the LibraryManagerTest folder.**

8 **Drag the ExampleLibrary into your Extensions folder.**

9 **Drag the MPW tool—LibraryManagerTest1—into your MPW Tools folder.**

10 **Open the TestTool folder.**

11 **Drag TestLibrary into your Extensions folder.**

12 **Drag the MPW tool—TestTool—into your MPW Tools folder.**

13 **If you want to use the debug version of the ASLM, drag the Shared Library Manager Debug file into your Extensions folder, and remove the Shared Library Manager extension that was placed in the Extensions folder by the Installer application.**

You should have only one Shared Library Manager file in the Extensions folder.

Installing the examples

The *ASLM Examples* disk contains a Read Me! file and source code examples. If you want to use these examples, copy them onto your hard drive.

Disk contents

This section describes the contents of the four ASLM disks.

ASLM Installer **disk**

This disk contains a variety of TeachText files, as well as the following files:

- the Installer application
- the Installer script
- the Shared Library Manager files folder which contains the Shared Library Manager extension and ASLM Resources file

ASLM Developer Tools **disk**

This disk contains a Read Me! file and three folders—Interfaces, Libraries, and Tools.

Interfaces folder

The Interfaces folder on the *ASLM Developer Tools* disk contains three folders:

- The CIncludes folder contains C and C++ header files that you need in order to develop C and C++ programs with the ASLM.
- The PInterfaces folder contains Pascal interface files that provide Pascal programmers with the interfaces needed to develop Pascal programs with the ASLM. See "Calling Shared Library Functions from Pascal" in Chapter 4 for more information on limitations when using the ASLM from Pascal.
- The RIncludes folder contains resource definition files. It also contains a `'libr'` resource template that you can use to decompile `'libr'` resources with the MPW tool DeRez. This resource template can help track down bugs that occur when function set, class, or library definitions are assigned improperly. It is described in Chapter 5, "Writing and Building Shared Libraries."

Libraries folder

The Libraries folder contains five libraries and a folder entitled THINK Libraries. The library files that end with the suffix .n.o are to be used with model near clients, and the library files that end with the suffix .o are to be used with model far clients. (For more information on the near and far memory models, refer to the latest MPW documentation and release notes.)

You can build your shared libraries to be used with either model near clients or model far clients, as explained in Chapter 4, "Writing and Building Clients." However, shared libraries are always created using the far memory model.

The Libraries folder contains the following files and folders:

- LibraryManager.o and LibraryManager.n.o are files that must be linked with all shared libraries and all applications that use shared libraries.
- LibraryManager.debug.o and LibraryManager.debug.n.o are debug versions of LibraryManager.o. and LibraryManager.n.o, respectively. They contain debugging symbols that may be useful when you debug your application or library. It is highly recommended that you use them during the development of your shared library or client application.

- TestLibrary.o is a file that you must link with your applications and shared libraries if you want to subclass the `TTestTool` class, which is used in the TestTool example program on the *ASLM Examples* disk.
- THINK Libraries (currently, support for THINK C/C++ 6.0 is only experimental).

  THINK C/C++ 6.0 users must use the libraries in the THINK Libraries folder when writing ASLM clients. Refer to the Read Me! file on the *ASLM Developer Tools* disk for the latest details regarding THINK C/C++ 6.0 support.

  THINK users should use LibraryManagerClient.o and LibraryManagerUtils.o when linking clients instead of LibraryManager.o, because LibraryManager.o contains references to routines that are only present if you are linking a shared library.

  The LibraryManagerClient.o and LibraryManagerUtils.o libraries do not include the routines that refer to the nonexisting routines. The routines that are removed are not needed by clients so clients will still be able to link.

  LibraryManagerUtils.o contains the client object files for the ASLM libraries that implement classes having a class ID that starts with `slm:supp$`. Like LibraryManager.o, there are also model near versions and debug versions.

Tools folder

The tools in the Tools folder include the following files:

- `BuildSharedLibrary` is an MPW script that builds shared libraries and client object files.
- `CreateLibraryLoadRsrc` is an MPW tool that lets clients and libraries create a resource that includes information about all the function sets and classes that they depend on so that they can be easily preloaded.
- `LibraryBuilder` is an MPW tool that is executed by the `BuildSharedLibrary` script and does most of the work when you build a shared library.
- `LinkSharedLibrary` is an MPW script that links shared libraries when you choose not to have `BuildSharedLibrary` do the linking for you.

ASLM Debugging Tools **disk**

The *ASLM Debugging Tools* disk contains a Read Me! file and the following tools and applications:

- The ASLM Debugger Prefs file contains MacsBug debugger macros and templates which are mainly used by ASLM engineers for debugging. You can put the contents of this file in the MacsBug Debugger Prefs file.
- The Inspector application, located in the Inspector folder, helps you inspect objects that are implemented in shared libraries, lets you see which function sets, classes, shared libraries, and shared library files the ASLM currently knows about, and provides some useful information about them. Appendix B, "ASLM Utility Programs," has further information on the Inspector application.

  The Inspector folder also contains the InspectorLibrary and WindowStackerLibrary files, which are used by the Inspector application.
- The TraceMonitor application creates a window where shared libraries and clients can send traces to help assist with debugging code (see Appendix B "ASLM Utility Programs" for more information).
- The LibraryManagerTest1 file, located in the LibraryManagerTest folder, is an MPW tool that performs a quick test of the ASLM. Appendix B, "ASLM Utility Programs," has further information on the LibraryManagerTest1.

  The LibraryManagerTest folder also contains the ExampleLibrary file which is used by LibraryManagerTest1 and LibraryManagerTest2.
- The TestTool file, located in the TestTool folder, is an MPW tool that is used to test certain ASLM functions and utility classes. Appendix B, "ASLM Utility Programs," has further information on TestTool.

  The TestTool folder also contains the TestLibrary file which is used by TestTool.
- The Shared Library Manager Debug file is a debug version of the ASLM. To use this version, drag it into the Extensions folder and drag the Shared Library Manager file out. Then reboot your machine. It is highly recommended that you use the debug version of the Shared Library Manager extension while developing your shared libraries and shared library clients. It contains code that will notice many developer errors and enter the MacsBug debugger with a message when it notices a problem.

ASLM Examples **disk**

The *ASLM Examples* disk contains a Read Me! file and seven folders containing source code example libraries and clients that help you learn how to develop and build ASLM clients and shared libraries.

- Example Tools
- ExampleLibrary
- FunctionSetInfo
- Inspector
- Sample Apps
- Sample INIT
- TestTools

Details of the contents of the ASLM Examples folder are given in Appendix C, “Using the Example Program.” Information on the ExampleLibrary, Inspector, and TestTool folders can also be found in Appendix B, “ASLM Utility Programs.”

Preparing to use the ASLM

With the ASLM installed, you can install any shared library by registering its library file or by simply dragging its library file into an appropriate folder (the System 7 Extensions folder, the System 6 System Folder, or a registered library file folder). Then, when an application needs to use the shared library, the ASLM dynamically loads and links the library.

The Shared Library Manager extension loads at boot time and stays loaded. When you have registered a library file or have dragged it into a library file folder, you do not have to reboot to use the shared libraries contained in the library file. It will be recognized immediately by the ASLM.

The operating system ordinarily loads the Shared Library Manager extension before it loads any other extensions. However, if you have installed the System 7 Tuner 1.1.1 and have AppleTalk turned off, the Shared Library Manager extension is not loaded and will not be usable. This behavior is caused by a feature in the System 7 tuner. It is corrected in system software version 7.1. If you do not have System 7.1, the workaround is to have AppleTalk turned on.

II Developing Clients and Shared Libraries

4 Writing and Building Clients

This chapter describes how to write a client, build a client, set up the current client, call shared library functions from C, C++, Pascal, and assembly languages, and create instances of classes that are implemented in shared libraries.

Overview

A program that makes use of shared libraries is called a client. Clients fall into three categories:

- an application or some other kind of code that has called `InitLibraryManager`
- a shared library
- the ASLM itself

Clients can use shared libraries that you write yourself, as well as the utility libraries supplied with the ASLM and by third party developers. This chapter explains how to write and compile clients, and how to dynamically link clients with shared libraries.

To develop your own shared libraries, you must have MPW 3.2 (or later) installed on your hard disk. For C++ development, you also need MPW C++ version 3.2 or 3.3. You must also copy a number of header files and tools as described in Chapter 3, “ASLM Installation.”

When you have set up your ASLM development system, you can write, compile, and link your own shared libraries and clients.

Each time you create a shared library, you must make it accessible to the clients that will use it. For information on how to make a shared library accessible to clients, see “Registering Shared Library Files and Folders” in Chapter 7, “ASLM Utilities,” or “Registering Shared Library Files” in Chapter 5, “Writing and Building Shared Libraries.”

Writing a client

When you write a client there are three basic rules to follow:

- Call `InitLibraryManager` in your client’s initialization section.
- Make sure your client either preloads the shared libraries it will use or uses exception handling to deal with shared libraries that may not exist or be loadable.
- Call `CleanupLibraryManager` before your client quits.

In the code between `InitLibraryManager` and `CleanupLibraryManager` calls, you can do just about anything that programs written in your language of choice can do, plus one thing that ordinary programs cannot: you can call functions and create classes that are implemented in shared libraries.

Before you can call a function implemented in a shared library, you must link the file that contains your source code with an object file provided by the developer of the shared library. This file, called a *client object file*, by convention has the extension .cl.o. It contains stubs for all the functions and classes implemented in the shared library. These stubs are responsible for loading the shared library and calling the implementation of the function or exported class non-virtual member functions. However, it does not contain the implementation of any of the routines implemented in the library. Note that virtual function calls are made through the object's vtable and present no additional overhead.

The following steps show how to write a shared library client in C that calls functions that are implemented in a function set in a shared library. The example assumes that the functions are contained in the function set whose id is `kCoolFunctionSetID` and the interfaces for the functions are contained in the header file "CoolLibrary.h". The `DoSomeThingGreat` and `DoSomeThingGreater` functions are both implemented in a shared library.

```
#include <LibraryManager.h>
#include <CoolLibrary.h>

// get ready to use the ASLM
if (InitLibraryManager(
      0,               /* we don't need memory in our local pool */
      kCurrentZone,    /* use application zone = current zone */
      kNormalMemory    /* default memory type, no VM stuff */
      ) == kNoError )
{
      // make sure that the shared library is loaded
      if (LoadFunctionSet(kCoolFunctionSetID) == kNoError)
      {
            // call some functions
            DoSomethingGreat();
            DoSomethingGreater();

            // call UnloadFunctionSet so the library can be unloaded
            UnloadFunctionSet(kCoolFunctionSetID);
      }

      // now we're all done using the ASLM
      CleanupLibraryManager();
}
```

`InitLibraryManager` is called before shared libraries or other ASLM facilities may be used. It creates the client's *local library manager* object, which is mainly used behind the scenes as the client's interface to the ASLM. `CleanupLibraryManager` is called when the client is finished using the ASLM. The "Creating and Deleting the Local Library Manager" section in Chapter 7 provides more information on the local library manager object and also describes `InitLibraryManager` and `CleanupLibraryManager` in detail.

The `LoadFunctionSet` call was made to make sure that the shared library was already loaded before attempting to call the functions implemented in the shared library. This prevents potential problems from arising if the shared library cannot be found or loaded. `LoadFunctionSet` and `UnloadFunctionSet` are explained in the "Loading and unloading shared libraries" section of Chapter 6. Exception handling could also be used instead of preloading the necessary shared libraries. Exception handling is explained in the "Exception handling" section of Chapter 6. `kCoolFunctionSetID` is a macro that defines the C string which is the function set id of the function set that the functions are in. This macro will always be located in the header file which declares the functions that you are using. Function set id's are explained in more detail in the "TFunctionSetID" section of Chapter 9.

When the `DoSomeThingGreat` function is called, what is actually called is a function stub that the client is statically linked with. This stub is responsible for calling into the ASLM to make sure that the shared library implementing `DoSomeThingGreat` is loaded and to store the address of the `DoSomeThingGreat` in the stub's cache. The stub can then call the actual implementation of `DoSomeThingGreat` . On subsequent calls to `DoSomeThingGreat`, the function address will already be cached with the stub so it can be called with just a few instructions.

After the client is finished calling the functions in the shared library, the client calls `UnloadFunctionSet` to undo the `LoadFunctionSet` call and then calls `CleanupLibraryManager`. When a client finishes using the ASLM, the client should always calls `CleanupLibraryManager`, although it will be called automatically for application clients.

As you can see, other than some initial setting up, calling functions in a shared library is no different than calling functions that the client is statically linked with.

Building a client

Figure 4-1 shows the steps required to build a client. The file Client.c is the source code for the client that is being built. C compiles Client.c, which includes the header file XXLibrary.h and builds an object code module named Client.c.o.

Once Client.c.o is built, it can be linked with the LibraryManager.o file and the client object file XXLibrary.cl.o. The result of the build is the client XXClient.

The XXLibrary is the shared library that the client will use. It must be registered with the ASLM in order to use it.

The file XXLibrary.h is a shared library header file; that is, an interface file that contains declarations of the functions and classes implemented in the shared library. XXLibrary.h is needed by the client to identify the functions and classes the library exports, and the interface to each.

The file XXLibrary.cl.o is the client object file; that is, a file that contains information including stubs for constructors, destructors, function sets, and any non-virtual methods that are exported by the shared library. The client object file must be linked with the client if the client calls any exported function or creates any objects implemented in the library. The exception to this is clients that use `NewObject` to create objects and `GetFunctionPointer` to call functions in function sets. These clients do not need to link with the client object file.

Two of the files shown in shadowed boxes—XXLibrary.h and XXLibrary.cl.o—are also needed to build shared libraries, as you will see later in Figure 5-1, “Building a Shared Library.” These two files are provided by the creator of the shared library.

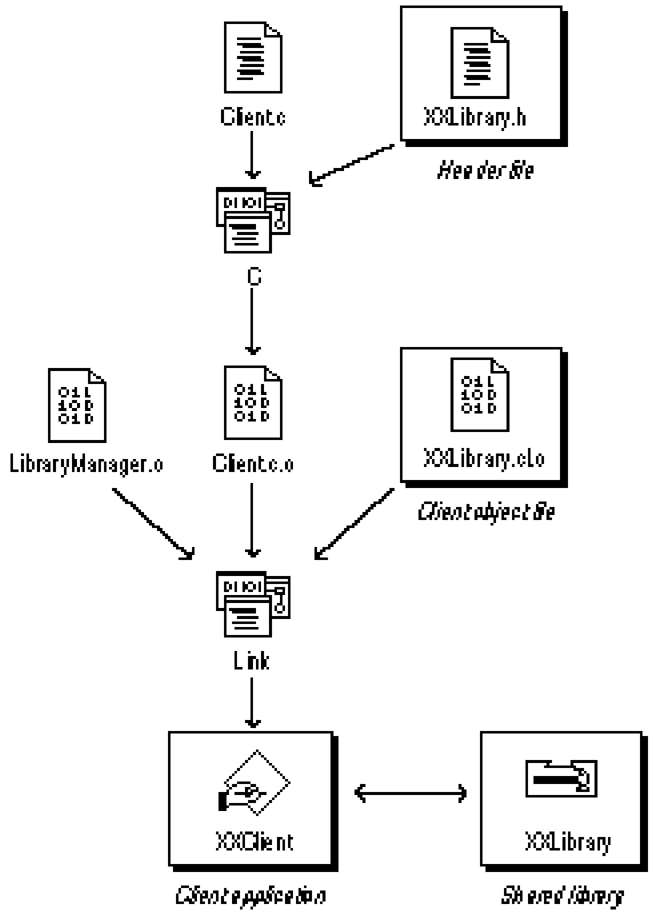


Figure 4-1 Building a client

Makefiles for building clients

You can learn how to write makefiles that build shared library clients by examining the makefiles for the example programs that are supplied with the ASLM. The example programs are located on the *ASLM Examples* disk and are discussed in Appendix C “Using the Example Programs.”

Refer to “Makefiles” in Chapter 5, “Writing and Building Shared Libraries,” for an example of a makefile that builds a client and an associated shared library.

Calling shared library functions from Pascal

You can call ASLM routines from Pascal in the same way that you call them from C. Pascal interface files are provided with the ASLM in the PInterfaces folder.

To call routines from Pascal, the writer of the shared library must provide a Pascal interface file (.p file) which contains the interface to functions exported by the shared library. This Pascal interface file can be provided instead of, or in addition to, the C-style include file (.h file), depending on whether the shared library writer also wants to support C programmers.

The PInterfaces folder contains the Pascal interface files LibraryManager.p and LibraryManagerUtilities.p, which provide Pascal programmers with all interfaces that C programmers have access to, with the following exceptions:

- Exception handling macros are not supported, but you can still call `Fail`. Because of this, all shared libraries to be used must be explicitly loaded first, otherwise the client runs the risk of throwing an exception when a shared library cannot be loaded for some reason. This will cause the application to quit.
- `Trace` is limited to one parameter: the string to output. No formatting is supported.
- `AtomicSetBoolean`, `AtomicClearBoolean`, and `AtomicTestBoolean` are not supported.
- All routines that take `StringPtr` parameters require the strings to be C strings, not Pascal strings. These routines are `Trace`, `GetSharedNamedResource`, `GetSharedResourceInfo`, `GetFunctionPointer`, and `Fail`.
- `DebugBreak` and related routines and macros are not supported.

Calling shared libraries from assembly language

Calling ASLM routines from assembly language does not introduce any particular problems, except for the usual issues that arise when you incorporate assembly-language code into programs written in other languages. Make sure you use C/Pascal register conventions.

Creating instances and calling member functions of shared classes

Using the ASLM from C++ is much the same as using it from C, except that there are a couple of different things to be aware of. The following code fragment shows how a C++ client can create objects that are instances of classes that are implemented in shared libraries, and how the client can call member functions of those objects. In this example, an instance of class named `TMyFirstClass` is created. `TMyFirstClass` is implemented in a shared library whose location the client need not be aware of. The interface for `TMyFirstClass` is located in the "CoolLibrary.h" header file.

```
#include <LibraryManager.h>
#include <CoolLibrary.h>

// declare a variable to point to our object
TMyFirstClass* first = NULL;

// get ready to use the ASLM
if (InitLibraryManager() == kNoError)
{
      // make sure that the shared library is loaded
      if (LoadClass(kTMyFirstClassID) == kNoError)
      {
            // create an object
            first = new TMyFirstClass;

            // call a method
            first->DoSomethingGreat();

            // delete the object
            delete first;

            // call UnloadClass so the library can be unloaded
            UnloadClass(kTMyFirstClassID);
      }

      // now we're all done using the ASLM
      CleanupLibraryManager();
}
```

Just as in the C example given earlier, the client must call `InitLibraryManager` before using the ASLM and call `CleanupLibraryManager` when finished using the ASLM. One difference in this example is that the C++ client was able to take advantage of the default parameters for `InitLibraryManager` and not explicitly pass any to it.

Also, as was done in the C example given earlier, the client had to make sure that the shared library to be used was loaded. This was done by calling `LoadClass` and passing in the class ID of the class that will be used. The class id is a C string the is declared in the class' header file using the macro k<classname>ID. Class id's are described in detail in the "TClassID" section of Chapter 9. `LoadClass` and `UnloadClass` are explained in the "Loading and Unloading Shared Libraries" section of Chapter 6. Exception handling could also be used instead of preloading the necessary shared libraries. Exception handling is explained in the "Exception Handling" section of Chapter 6.

After this client has called `InitLibraryManager` and `LoadClass` , an instance of `TMyFirstClass` is created using the `new` operator and the constructor for the class. The class's constructor accepts parameters that are passed in normal C++ fashion.

The constructor stub for each class is statically linked with the client. The first time a stub is called to construct an object, it calls the ASLM which takes care of loading the library if it is not loaded already and placing the address of the constructor in the constructor stub's function cache. The stub can then call the constructor. Each subsequent time the stub is called, it can directly jump to the constructor after first checking that the library was not unloaded. The constructor increments the use count for the class each time it is called. The destructor for each class then decrements the use count so that the library can be unloaded when all objects in a given library have been deleted.

After the client has created an instance of `TMyFirstClass`, a member function is called and then the object is deleted. Finally, the client calls `UnloadClass` to undo the `LoadClass` call and then calls `CleanupLibraryManager`. When a client finishes using the ASLM, the client should always calls `CleanupLibraryManager`, although it will be called automatically for application clients.

As you can see, clients can create the objects and call their methods in ordinary C++ fashion. The only extra conventions that the client must observe are calling `InitLibraryManager` and `CleanupLibraryManager` and also either preloading shared libraries to be used or use exception handling in order to deal with shared libraries that are either missing or are not loadable.

More information on creating instances of shared classes and calling methods is discussed in Chapter 6. Topics include creating static objects and stack objects, using `NewObject` to create an object with a given class ID, using the ASLM global `new` and `delete` operators, and the advantages that virtual functions have over non-virtual functions.

The current client

The current client is generally the currently executing application, but other clients (such as shared libraries and stand-alone code resources) have the ability to make themselves the current client also. The current client is generally used to determine on whose behalf something is done, such as allocating memory, opening a file, or setting up or making a callback.

The current client is important for a number of reasons. When a client or a shared library built with the `memory=client` option allocates memory using the default C++ `new` operator, the memory is allocated from the current client's local pool (also called the client pool). Also, when an exception is raised, the ASLM uses the current client's exception handling chain to determine who should catch the exception. When a library file is opened by calling `PreFlight` or `OpenLibraryFile`, the file is opened for the current client. Lastly, the ASLM per client data facility relies on the setting of the current client when deciding which client data "context" to return when a shared library calls `GetClientData`.

For these reasons, it is not generally safe to make a call into a shared library or into the ASLM unless the current client is defined.

Who needs to set the current client?

Any code that makes a call into a shared library or into the ASLM is responsible for making sure that the current client is set properly unless special arrangements have been made with the shared library so that it can handle being called with an invalid current client. The current client is invalid if the currently executing application is not an ASLM client and the current client was not explicitly set.

Normally, when an application client is executing, it is also the current client and does not have to do anything special to make sure that the current client is set properly unless it is called asynchronously. If the application client is called asynchronously, the current client may not be set properly. In this case, it is up to the application client to make sure it is set properly before the client calls a shared library or the ASLM. Setting up the current client within a routine that handles asynchronous events is usually handled by making sure the client that it wants set as the current client is passed to the routine, which can then call the `SetCurrentClient` function, which is described later in this section.

Shared libraries may want to change the current client so that default C++ memory allocations are made from the shared library's local pool rather than from the local pool of whoever is the current client when the library is called. Also, a shared library may have to set up the current client because it was called from code that did not set the current client to a valid client; for example, if the code in the shared library is an interrupt service routine or an I/O completion routine.

Determining the current client

The default setting of the current client is determined by the setting of the Macintosh low-memory global `CurrentA5`. `CurrentA5` is always set to the current value of the global world of an application. Ordinarily, it is set to the global world of the currently executing application. Therefore, the application is normally also the current client. If the current client makes a call into a shared library, the setting of `CurrentA5` global does not change so the application remains the current client.

Setting the current client

The ASLM provides several functions used to override the setting of the `CurrentA5` low-memory global, making it possible to specify the current client. These functions do not change the value of `CurrentA5`, but rather tell the ASLM to use a client other then the one specified by the setting of `CurrentA5` as the current client. They include the following:

- `SetCurrentClient` makes the client passed as a parameter the current client.
- `SetSelfAsClient` makes the client issuing the call the current client.
- `SetClientToWorld` makes the owner of the current global world the current client.

The ASLM also provides the function `GetCurrentClient`, which returns the current client. It is useful for getting and then saving the current client so you can set up the current client sometime in the future. For example, suppose your shared library is some sort of driver that may need to notify one of its clients of an event at interrupt time. When the client is "setup," the driver can call `GetCurrentClient`. When the client needs to be notified, the driver can then call `SetCurrentClient`. Thus the client does not need to set the current client.

This is the syntax of these four calls:

```
TLibraryManager*    GetCurrentClient(void);
TLibraryManager*    SetCurrentClient(TLibraryManager*);
TLibraryManager*    SetSelfAsClient(void);
TLibraryManager*    SetClientToWorld(void);
```

The `GetCurrentClient` function returns the current client. The `SetCurrentClient`, `SetSelfAsClient`, and `SetClientToWorld` functions all return the previous current client. This client should be passed to `SetCurrentClient` to restore the current client.

The current client is represented by the client's local library manager, and, therefore, all of the current client routines return a `TLibraryManger*` object, and `SetCurrentClient` accepts as a parameter the `TLibraryManager*` belonging to the client to be set as the current client.

You can also use `EnterSystemMode` to change the current client. It sets the ASLM as the current client. The `LeaveSystemMode` function restores the current client. For more information on `EnterSystemMode`, see "Entering and Leaving System Mode" in Chapter 7, "ASLM Utilities."

The `GetClientPool` function crashes when the `CurrentA5` low-memory global does not belong to a valid ASLM client and the current client has not been set. Usually, `GetClientPool` is not called directly, but is called automatically when you create an object that is implemented in a library built with the `memory=client` option. The `GetClientPool` function can crash when a non-ASLM client invokes code that is in an ASLM client. For example, if you implement a HyperCard XCMD as an ASLM client, the XCMD should call `SetSelfAsClient` immediately after it calls `InitLibraryManager`, and should restore the current client immediately before it calls `CleanupLibraryManager`. Otherwise, the ASLM considers HyperCard the current client. A crash might then occur if the XCMD tries to create objects or allocate memory.

WARNING If `SetCurrentClient`, `SetSelfAsClient` or `SetClientToWorld` is called, it is necessary to restore the current client before returning from the routine that set the current client. It is imperative that if an application sets the client, it restores the current client before calling `EventAvail`, `GetNextEvent`, or `WaitNextEvent`. This also means that any other client or shared library that sets the current client should restore the current client before calling any routine that may result in `EventAvail`, `GetNextEvent`, or `WaitNextEvent` being called. Normally applications need to set the client only in callbacks (completion routines, operation process procs, notifier notify procs, and so on) that use the ASLM.

In the debug version of the ASLM, you will enter MacsBug with a warning if the current client is not set to `NULL` when `EventAvail`, `GetNextEvent`, or `WaitNextEvent` is called. This is done because if the current client is not set to `NULL` when calling one of the above traps, problems can occur. For example, suppose Client A is an ASLM client that leaves the current client unset (so it will always be the current client when it is running) and then calls `WaitNextEvent`. Client B takes over, sets the current client, and calls `WaitNextEvent`. Client A then regains control, but is no longer the current client because Client B left it set to another client. If you know (or think) the current client has been set and want to call one of the above traps, do the following:

```
TLibraryManager* savedClient = SetCurrentClient(NULL);
WaitNextEvent();            // or EventAvail or
GetNextEvent
SetCurrentClient(savedClient);
```

You can use this same technique when calling a routine that calls one of the above traps. In fact, if the routine you are calling knows nothing about ASLM, it is the caller’s responsibility to make sure the current client is set to `NULL`.

The LibraryManager.o file

The LibraryManager.o file illustrated in Figure 4-1 is an MPW library file supplied for ASLM client and library developers. It contains

- client object file code (.cl.o code) for shared libraries supplied with the ASLM
- routines defined in the ASLM header files (the client will dynamically link with most of these routines)
- other behind-the-scenes routines that are used internally

The LibraryManager.o file should be linked with all clients before any C libraries are linked. It should also be linked before CPlusLib.o unless you want to use the global `new` operator supplied by CPlusLib.o. See “Using the ASLM Global `new` and `delete` Operators” in Chapter 6, “Using the ASLM,” for more details.

The LibraryManager.n.o file is similar to the LibraryManager.o file, except that it is meant only for model near clients and, therefore, is not compiled with model far. LibraryManager.debug.o and LibraryManager.debug.n.o are debug versions of the library files and contain debugger breaks and MacsBugs symbols useful when trying to debug clients and shared libraries.

5 Writing and Building Shared Libraries

This chapter describes how to write and build shared libraries, create symbol files, use makefiles, and write exports files. It also discusses related topics that you need to consider when creating shared libraries.

Overview

Before you can build a shared library, you need at least three source files:

- One or more source files that contain the implementation of your library's classes and functions. The files that contain your library's implementation can be written in any language that is compatible with MPW, such as C++, C, Pascal, or assembly language.
- A header (or Pascal interface) file that provides declarations for the functions and classes that your library will export. You may also have one or more private header files for declarations that the user of your library will not need. Header files written in C format always have filenames that end with the suffix .h. Pascal interface files can end with the suffix .p.
- An export definition file (also called an exports file or .exp file) that defines classes and function sets that are to be exported from a shared library. An export definition file is always written in C-language style and always has a filename ending in the suffix .exp.

When you have written the source files that are needed to create a shared library, you must write and execute a makefile that compiles your source files into object code files from which a shared library can be built.

For an example of a shared library makefile, see "Makefiles" later in this chapter.

Writing a shared library

To write a shared library, you do not need to do anything special with the source code. However, you do need to create an exports file, as described in "Writing an .exp File" later in this chapter.

You can also use any of the many utility functions and classes that are supplied with the ASLM to add extra power and functionality to your programs. The sample programs in the *ASLM Examples* disk demonstrate what you can do with the collection of ASLM utility functions and classes.

After you have written the source files, you must use the proper tools to build the shared library. This is described in the next section.

Building a shared library

Figure 5-1 illustrates the process of building a shared library. To build a shared library, you must provide two input files to the `BuildSharedLibrary` script (the library builder):

- An *input object file* (an object file named with the suffix .o) from which a shared library can be created. In Figure 5-1, the input object file is created by compiling XXLibrary.c, which includes XXLibrary.h. When you have multiple source files that make multiple object files, you can use the MPW `Lib` command to create one input object file.
- An exports file, called XXLibrary.exp in Figure 5-1, which contains important information about a particular shared library, including a special kind of declaration called a *library definition.* A library definition usually contains the library's library ID and version number, along with other kinds of information about the library—for example, information about the pool from which the shared library allocates memory. More information about the exports file is provided in "Writing an .exp File" later in this chapter.

The two files in Figure 5-1, XXLibrary.h and XXLibrary.cl.o, are files that are also needed to build a client. These two files also appear in Figure 4-1, "Building a Client," in Chapter 4, "Writing and Building Clients."

The XXLibrary.c file is the source code for the implementation of the library. The XXLibrary.h file is the same header file shown earlier in Figure 4-1. A shared library header file is an interface file that contains the declarations for classes and functions exported by the shared library. This file is used by both the shared library source files and the client's source file.

The XXLibrary.cl.o file shown in Figure 5-1 is the same client object file shown earlier in Figure 4-1. The client object file is a file that contains the stubs that will dynamically link clients to your shared library. The client object file must be linked with the client if the client calls any exported function or creates any objects implemented in the library. The exception to this is clients that use `NewObject` to create objects and `GetFunctionPointer` to call functions in function sets. These clients do not need to link with the client object file. Also, a shared library must link with its own client object file if it exports any classes.

When you build a shared library, two output files are generated; a *shared library file* and a *client object file*:

- The shared library file that is produced during the build process is an actual shared library that can be placed in the Extensions folder. The shared library file is always built using model far.
- Client object files are files to which clients of a shared library must link. A client object file usually has a filename that ends with the suffix .cl.o or .cln.o, depending on whether the clients linking with your library will be model near clients (.cln.o) or model far clients (.cl.o). You must build a model far client object file if you want to let model far clients use your library, or if your library exports C++ classes. You must build a model near client object file if you want to let model near clients use your library.

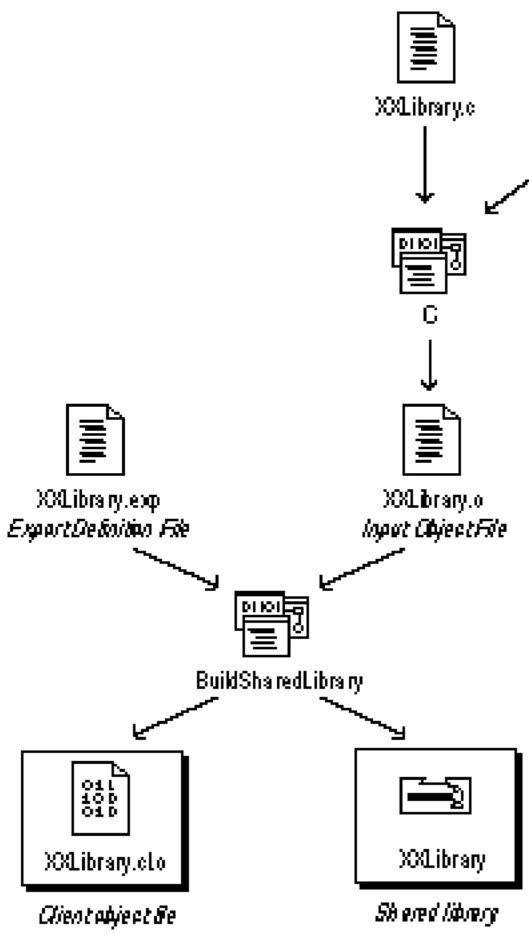


Figure 5-1 Building a shared library

Build utilities

The Tools folder on the *ASLM Developer Tools* disk contains four utilities that you use to build a shared library: two MPW Tools and two MPW scripts. Before using these utilities, the two scripts must be placed in the MPW scripts folder and the two tools in the MPW tools folder. These are the four utilities:

- The `BuildSharedLibrary` script is an MPW script that builds a shared library and a client object file from an input object file and an exports file. The `BuildSharedLibrary` script calls the `LibraryBuilder` tool.
- The `LinkSharedLibrary` script is an MPW script that links a shared library. You can choose to use `LinkSharedLibrary` or `BuildSharedLibrary` to do your linking. You must use the `LinkSharedLibrary` script when you want to build two or more shared libraries with circular dependencies (libraries that depend on each other's client object files).
- The `LibraryBuilder` tool is an MPW tool that is executed by the `BuildSharedLibrary` script. The `LibraryBuilder` tool creates an interim script that is used in the build process, and also creates an interim file called an *initialization file*. It uses these files, along with the input object file and the exports file that you provide, to create a shared library file and client object files. The `LibraryBuilder` tool does most of the work when you build a shared library.
- The `CreateLibraryLoadRsrc` tool is an MPW tool that allows ASLM clients and libraries to create a resource that contains information about the function sets and classes they depend on. The `LoadLibraries` routine uses this resource to preload all libraries on which a client is dependent.

Using `BuildSharedLibrary`

To build a shared library, you should make sure that all the modules in the object file you are using were built using model far. Then make sure that the `LibraryBuilder` tool is placed in your MPW Tools folder, and run the `BuildSharedLibrary` script.

The syntax of the `BuildSharedLibrary` command is:

```
BuildSharedLibrary InputObjectFile [-y ScratchPath]
      -exp InputExportFile [-far OutputFarClientObjectFile]
      [-near OutputNearClientObjectFile] [-macsbug]
      [-privateFar OutputFarPrivateFile]
      [-privateNear OutputNearPrivateFile] [-lib LibraryObjectName]
      [-obj OutputObjectBaseName] [-restype codeResourceType]
      [-resid n] [-thinkC] [-map MapFileName] [-sym SymbolOption]
      [-symfile SymFileName] [-w1] [-w2] [-w#] [-p] [-v] [-c] [-e]
      [-help] [-noMerge] [-noVirtualExports] [-keepClientFiles]
      [-i IncludePath] [ObjectFilesToLinkWith...]
      [-link LinkerOptions] [-logout OutputLogFileName]
      [-log InputLogFileName] [-dolog]
```

where:

`InputObjectFile`

The first parameter on the command line that is not preceded by a hyphen (`-`) is the name of the *input object file*—that is, an object file (which may be the output of an MPW `Lib` command) that you want to convert into a shared library. The input file itself is not affected by this command. This parameter is mandatory.

It is best for this object file to include only the implementation of classes and functions you export. Other routines that the exported classes and functions depend on can be placed in object files specified with the `-link` or `ObjectFilesToLinkWith` parameters. Although this procedure is not required, it will help speed up builds and you should definitely avoid using `Lib` to combine LibraryManager.o with the *`InputObjectFile`*.

`-y` *`ScratchPath`*

This optional parameter specifies the path name of a scratch folder for all temporary files created during the build process. If you do not specify a scratch path, the `BuildSharedLibrary` script places scratch files in the folder specified by the MPW variable `{TempFolder}`. If no `{TempFolder}` variable is defined, the `BuildSharedLibrary` script uses the path name specified by the MPW variable `{CPlusScratch}`. If neither of these variables is defined, scratch files are placed in the current directory.

`-exp` *`InputExportFile`*

This mandatory parameter specifies the name and path of your exports file. Normally, this file is named *LibraryName*.exp.

`-far` *`OutputFarClientObjectFile`*

This optional parameter allows you to link model far clients with your shared library. The `-far` parameter specifies the path name of the client object file that is generated by the build process (a file with a name that ends with the suffix .cl.o). If your library exports C++ classes, you must use the far parameter and link your shared library with the model far client object file that is created. Otherwise, link errors are generated. Specifically, references to constructors and destructors of exported classes will be unresolved. You can use the `-far` option and the `-near` option in the same command.

Note: `-near` and `-far` merely specify the kinds of clients that can link with your library. They do not affect the library itself; shared libraries are always built using model far.

`-near` *`OutputNearClientObjectFile`*

This optional parameter allows you to link model near clients with your shared library. The `-near` parameter specifies the path name of your client object file (a file with a name that ends with the suffix .cln.o). Model near clients of your shared library must link with this file. You can use the `-near` option and the `-far` option in the same command.

Note: `-near` and `-far` merely specify the kinds of clients that can link with your library. They do not affect the library itself; shared libraries are always built using model far.

`-macsbug`

This optional parameter places MacsBug symbols in the client object file. It is useful when you are trying to debug your shared library or client. Stubs for the exported routines will have MacsBug symbols that start with `stub_`. The debug versions of LibraryManager.o are built in this manner.

`-privateFar` *`OutputFarPrivateFile`*

This optional parameter specifies an output object file for model far private stubs. For more information, see the description of the `private=` option for the `Class` and `FunctionSet` declarations in “Writing an .exp File” later in this chapter.

`-privateNear` *`OutputNearPrivateFile`*

This optional parameter specifies an output object file for model near private stubs. For more information, see the description of the `private=` option for the `Class` and `FunctionSet` declarations in “Writing an .exp File” later in this chapter.

`-lib` *`LibraryName`*

This optional parameter specifies the name and path of the shared library file that the build process produces. If `-lib` is missing, `BuildSharedLibrary` creates only the client object files, and you need to invoke the `LinkSharedLibrary` script later in order to actually create the shared library. This mode is useful when you have two or more shared libraries that are interdependent.

This library file normally contains two resource types: a `'libr'` resource, which contains a dictionary of the classes and function sets that your library exports, and your library's actual code segment resources (normally `'code'` resources). It may also contain a third resource type: a `'libi'` resource, which contains a map of all of the function sets and classes on which your library depends. If your library has no external dependency, this resource is missing.

The `BuildSharedLibrary` command creates only one shared library file at a time, but if you use different resource types for the code segments in your library, and unique numbers for your `'libr'` resources, you can use the MPW Rez tool to `rez` multiple libraries together into a single library file. (For more information related to this topic, see the `-restype` and `-resid` parameters, below).

`-obj` *`ObjectFileBaseName`*

This optional parameter specifies the name and path for intermediate files. If this parameter is missing, the intermediate files are deleted once the library is created. The advantage of using this parameter is that if one of the client files that your library depends on changes, you only need to relink the library. However, if these files are not available, `BuildSharedLibrary` must do a complete rebuild of your library, which takes a longer time. You never need to deal with these files directly. They are only used by `BuildSharedLibrary` and `LinkSharedLibrary`.

There are four intermediate files created, and they are named by appending the following extension to your *`ObjectFileBaseName`*:

.lib.o	A copy of *InputOjbectFile* with some module names changed
.libr.r	Resource to `rez` with the shared library including the `'libr'`, `'libi'`, and library code resources
.deps	File used to create the `'libi'` resource
.init.o	Initialization code for the library

`-restype` *`codeResourceType`*

This optional parameter allows you to specify a resource type for your shared library's code resources. The default code resource type of a shared library is '`code`'. The `-restype` parameter is useful only if you plan to `rez` multiple libraries together into a single library file.

`-resid` *`n`*

With this optional parameter, you can give your shared library's `'libr'` resource a resource ID number. The default resource ID number of a `'libr'` resource is 0. The `-resid` parameter is useful only if you plan to `rez` multiple libraries together into a single library file. This resource ID is also used if a `'libi'` resource is generated for the library.

`-thinkC`

This optional parameter specifies that the *`InputObjectFile`* was compiled with the Symantec C or C++ compilers for MPW.

`-map` *`MapFileName`*

This optional parameter generates a linker map file. It must be passed as a `LinkSharedLibrary` parameter and not as a linker parameter in the `-link` section. This parameter is only used if you also specify the `-lib` parameter. Otherwise use it with `LinkSharedLibrary` instead.

`-sym` *`SymbolOption`*

This optional parameter causes symbols to be placed in the symbol file specified with the `-symfile` option.

`-symfile` *`SymFileName`*

This optional parameter causes any SYM file created by linking the shared library to be copied to the specified path.

`-w1`, `-w2`, and `-w#`

These optional parameters are used to specify the level of warning you want produced.

`-p` (or `-progress`)

This optional parameter causes the `BuildSharedLibrary` script to run in a progress mode, generating a brief progress report. It is useful for debugging build problems.

`-v` (or `-verbose`)

This optional parameter turns on a verbose mode during the build process. The verbose mode provides more detailed progress information than the progress mode. The report generated in verbose mode lists the names of classes and global functions that were not exported. It is useful for debugging build problems.

`-c`

This optional parameter informs the `LibraryBuilder` that your object files contain no code written in C++. This parameter forces `BuildSharedLibrary` to match function names exactly when function names contain two consecutive underscore characters (`__`). C users can always safely use this parameter, but they only need to use it if a function to be exported contains two consecutive underscore characters.

This option is needed because normally `BuildSharedLibrary` only does partial matching of function names up to the first occurrence of two consecutive underscore characters. This is because C++ mangles function names so parameter information can be encoded in the function name. Mangled function names always start with the normal function name followed by two consecutive underscore characters and then the encoded parameter information. When the -c option is not used, `BuildSharedLibrary` only compares the part of the function name before the two consecutive underscore characters.

`-e`

This optional parameter forces `BuildSharedLibrary` to completely rebuild the library. By default, `BuildSharedLibrary` checks whether the modification date of the object file has changed since the library was last built, and does not reprocess the object file if the modification date has not changed.

`-help`

This optional parameter outputs a detailed list of all of the options to `BuildSharedLibrary`.

`-noMerge`

This optional parameter prevents the link of the shared library from merging all of the segments used by the MPW (or Symantec C/C++) libraries into the Main code segment.

`-noVirtualExports`

Use this optional parameter if you do not want stubs generated for virtual functions. This is easier than changing all your class export declarations to include `flags=noVirtualExports`. You can still explicitly export some virtual functions by using `exports=`. Also, you will still be able to make virtual function calls through the object's v-table.

`-keepClientFiles`

This optional parameter ensures that `BuildSharedLibrary` does not change the modification date of client objects files if their contents have not changed. It is explained in more detail in "Speeding Up Builds" later in this chapter.

`-i` *`IncludePath`*

This optional parameter, which can occur multiple times on the command line, supplies directory path names where the `BuildSharedLibrary` script should search for files that you have included in your exports file using the `#include` directive. You must provide a separate `-i` option for each search path you specify.

`ObjectFilesToLinkWith`

When all parameters that start with hyphens (`-`) have been evaluated, any other words that appear on the command line are assumed to be the names of object files. The first filename that appears on the command line is assumed to be the input object file. All other filenames are assumed to be the names of object files that must be linked with your shared library. For more information related to this topic, see the *`InputObjectFile`* entry earlier in this list.

`-link` *`LinkerOptions`*

This optional parameter causes everything that appears after it to be passed verbatim to the `Link` command that links your shared library. The `-link` parameter can be useful when you want to pass commands on to `Link`, such as commands to merge segments.

When `BuildSharedLibrary` links your shared library, it automatically merges all code segments used by MPW libraries into the Main code segment. If this is not what you want, you can override this feature by specifying a linker option with the `-link` option.

`-logout` *`OutputLogFileName`*

The `logout` switch specifies the output log file. The output log file is an ASCII text file that shows where various functions, v-tables, and so on, are being exported.

`-log` *`InputLogFileName`*

The `log` switch specifies an input log file. The log file is used to control the generation of the new library.

`-dolog`

The `dolog` switch actually enables the logging operations. (This is so that you can specify `-logout` or `-log` in your makefile, but nothing is done until you alias `BuildSharedLibrary` to be `BuildSharedLibrary -dolog`, or something similar.)

Building a shared library with circular dependencies

If you want to build a shared library that has a circular dependency with another library, you cannot build your shared library until you have created the client object file of the other shared library, and you cannot build the other shared library until you have created the client object file from the first library. (A circular dependency exists when there are two or more shared libraries that depend on each other's client object files.)

To build shared libraries with circular dependencies, you must split the build of your shared library into two phases. The first phase creates all the client object files that the build process requires. The second phase links the shared libraries.

Creating client object files and intermediate files

To carry out the first phase, you need to run the `BuildSharedLibrary` script as you normally would except omit the `-lib`, `-link`, and *`ObjectFilesToLinkWith`* parameters. This will create the object files and some intermediate files that will be needed to link the shared library, but it does not link the shared library. Intermediate files are described with the `-obj` option in the previous section.

Linking the shared library

After you create the client object files and intermediate files, you must run the `LinkSharedLibrary` script to link your shared library.

The syntax of the `LinkSharedLibrary` command is:

```
LinkSharedLibrary -lib LibraryName -obj InputObjectBaseName
      [-symfile SymFileName]  [-map MapFileName]
      [-noMerge] ObjectFilesToLinkWith... [-link LinkerOptions]
```

The `-link`, `-map`, `-noMerge`, and *`ObjectFilesToLinkWith`* parameters are the same as for `BuildSharedLibrary`. These are descriptions of the options and parameters that you can place on the `LinkSharedLibrary` command line:

`-lib` *`LibraryName`*

This parameter specifies the name and path of the shared library to be built and is the same as the `-lib` in `BuildSharedLibrary`. Either specify `-lib` with `BuildSharedLibrary`, in which case you will not be using `LinkSharedLibrary`, or omit it from `BuildSharedLibrary` and specify it with `LinkSharedLibrary`.

`-obj` *`InputObjectBaseName`*

This parameter must be the same as the file specified by the `-obj` *`ObjectFileBaseName`* parameter of the `BuildSharedLibrary` command.

`-symfile` *`SymFileName`*

This optional parameter specifies where to put the .SYM file. You must also use `-sym on` or `-sym on,nolines` in the `-link` section. For more information see “Creating Symbol Files” later in this chapter.

Creating symbol files

The `BuildSharedLibrary` and `LinkSharedLibrary` commands use the switch, `-symfile` *`SymFileName`*. If your link creates a .SYM file, it will be copied to the file *`SymFileName`*.

If you are using `BuildSharedLibrary` to link your library, you must also pass `-sym on` or `-sym on,nolines` to `BuildSharedLibrary`. Do not pass the `-sym` option to the linker by including it after the `-link` option. You should also use the `-symfile` option with `BuildSharedLibrary`.

If you are using `LinkSharedLibrary` to link your library then you should pass `-sym on` or `-sym on,nolines` after the `-link` option and include it with the options passed to `BuildSharedLibrary`. You also need to use the `-symfile` option with `LinkSharedLibrary`, but not `BuildSharedLibrary`.

For an example of creating a .SYM file using just `BuildSharedLibrary`, look at the ExampleLibrary makefile. For an example of creating a .SYM file using `LinkSharedLibrary`, look at the Inspector makefile. In both cases the MPW `{SymbolOption}` variable should be set to `-sym on` or `-sym on,nolines` to produce a .SYM file. It will not create one by default. Notice that there is no harm in using the `-symfile` option even if you are not going to produce a symbol file.

Makefiles

You can learn how to write makefiles that build shared libraries and clients by examining the makefiles for the example programs that are supplied with the ASLM. The example programs are provided in a number of folders as described in Appendix C, “Building Examples.”

A makefile example

Listing 5-1 is a makefile that builds a client named CSample and an associated shared library named CSampleLibrary. You can find the makefile in the CSample folder inside the Sample Apps folder. The makefile builds the shared library and its client from source files named Sample.h, SampleLibrary.h, Sample.c, SampleLibrary.c, Sample.r, SampleLibrary.exp, and SampleLibrary.r.

Listing 5-1 Makefile for the sample client and its shared library

```
#-------------------------------------------------------------------------------------
#       File:          Makefile
#
#       Contains:      This makefile creates CSampleLibrary and its client
#                      application called CSample.
#
#       Build Command: BuildProgram CSample
#
#       Copyright:     © 1993 by Apple Computer, Inc., all rights reserved.
#
#

SRC     = :Sources:
OBJ     = :Objects:
BLT     = :Built:

SLMCIncludes  = {SLMInterfaces}CIncludes:
SLMRIncludes  = {SLMInterfaces}RIncludes:

#-------------------------------------------------------------------------------------
#       TARGETS
#-------------------------------------------------------------------------------------

TARGETS        =       "{OBJ}SampleLibrary.cl.o" ∂
              "{BLT}CSampleLibrary" ∂
              "{BLT}CSample"

#-------------------------------------------------------------------------------------
#       DEFAULT RULES
#-------------------------------------------------------------------------------------

.c.o    ƒ      .c
       Echo "∂t∂tCompiling {Default}.c"
       C {DepDir}{Default}.c -o {Targ} {COptions}

#-------------------------------------------------------------------------------------
#       COMPILER/ASSEMBLER OPTIONS
#-------------------------------------------------------------------------------------

AOptions       =       -model far -case on
COptions       =       -model far -i {SRC} -mbg on -sym full,nolines -mf -b2 -opt full ∂
                       -i "{SLMCIncludes}"
```

Continued on following page ▶

```
#-----------------------------------------------------------------------------------
#       DEPENDENCIES
#-----------------------------------------------------------------------------------
"{OBJ}"        ƒ       "{SRC}"
CSample        ƒ       {TARGETS}
#-----------------------------------------------------------------------------------
#       CREATE SAMPLE SHARED LIBRARY
#-----------------------------------------------------------------------------------
"{OBJ}SampleLibrary.cl.o" ƒ "{OBJ}SampleLibrary.RSRC"
       SetFile -m . {Targ}
"{OBJ}SampleLibrary.RSRC" ƒ "{OBJ}SampleLibrary.c.o" "{SRC}SampleLibrary.exp"
       BuildSharedLibrary ∂
              {OBJ}SampleLibrary.c.o ∂
              -macsbug ∂
              -lib "{OBJ}SampleLibrary.RSRC" ∂
              -obj "{OBJ}CSampleLibrary" ∂
              -far "{OBJ}SampleLibrary.cl.o" ∂
              -exp "{SRC}SampleLibrary.exp" ∂
              -i "{SRC}" -i "{SLMCIncludes}" -i "{CIncludes}" -p ∂
              "{SLMLibraries}LibraryManager.o" ∂
              "{Libraries}Runtime.o"
"{BLT}CSampleLibrary"         ƒ {SRC}SampleLibrary.h {OBJ}SampleLibrary.c.o
 {OBJ}SampleLibrary.RSRC
       Echo "∂t∂tRezzing {Targ}"
       Rez -t libr -c OMGR -s "{OBJ}" ∂
              -i "{SLMRIncludes}" -i "{SRC}" ∂
              -o {Targ} "{SRC}SampleLibrary.r"
       SetFile -a ib {Targ}
#-----------------------------------------------------------------------------------
#       CREATE SAMPLE APPLICATION(CLIENT)
#-----------------------------------------------------------------------------------
"{BLT}CSample"       ƒƒ {SRC}Sample.h {OBJ}Sample.c.o {OBJ}SampleLibrary.cl.o
       Echo "∂t∂tLinking {Targ}"
       Link -w -model far ∂
              "{OBJ}Sample.c.o" ∂
              "{SLMLibraries}LibraryManager.o" ∂
              "{Libraries}Runtime.o" ∂
              "{Libraries}Interface.o" ∂
              "{OBJ}SampleLibrary.cl.o" ∂
              -o {Targ}
       SetFile {Targ} -t APPL -c 'MOOS' -a B
```

```
"{BLT}CSample"       ƒƒ {SRC}Sample.h {SRC}Sample.r {OBJ}Sample.c.o
{OBJ}SampleLibrary.cl.o
       Echo "∂t∂tRezzing {Targ}"
       Rez -i "{SRC}" -rd -o {Targ} "{SRC}"Sample.r -append

"{OBJ}SampleLibrary.c.o" ƒ "{SRC}SampleLibrary.h"
"{OBJ}Sample.c.o"    ƒ "{SRC}SampleLibrary.h" "{SRC}Sample.h"
```

Makefile example contents

Here is a list of the contents of each file that the makefile in Listing 5-1 processes:

- *Sample.h* contains declarations for the sample application.
- *SampleLibrary.h* contains the declarations of the functions exported by the shared library.
- *Sample.c* contains client source code.
- *SampleLibrary.c* contains the implementations of functions exported by the shared library.
- *SampleLibrary.exp* contains the library definition for the shared library, and the definitions of any function sets that are exported.
- *SampleLibrary.r* is the resource definition file for resources used by the shared library.
- *Sample.r* is the resource definition file for resources used by the client.
- *SampleLibrary.RSRC* is the compiled and linked implementation of the shared library.
- *CSampleLibrary* is the shared library that is placed in the Extensions folder. CSampleLibrary contains the resources in SampleLibrary.RSRC.
- *CSample* is the client application that uses CSampleLibrary.

Executing a shared library makefile

To execute a shared library makefile, execute the following command from the directory of the makefile:

```
make -f makefilename > make.out
make.out
```

Writing an .exp file

This section explains how to write the export definition (.exp) file needed to create a shared library. The .exp file defines any classes and function sets that you want to export from your shared library.

An exports file can contain comments (written in C-language comment style), `#include` directives, `#define` directives, a `Library` declaration, and any number of `FunctionSet` and `Class` declarations.

`Library` declaration

The `Library` declaration in a shared library's exports file contains important information about the library, including the library's library ID and the library's version number. It can also contain other parameters for configuring the library.

The following code fragment is an example of a `Library` declaration:

```
#define kLightLibID "appl:sample$TrafficLight,1.1"
Library {
      id = kLightLibID;
      version = 1.0b1;
      memory = client;
};
```

Syntax

The syntax of a full `Library` declaration is:

```
Library
{
     id = <LibraryIDString>;                  /* required*/
     version = <LibraryVersion>;              /* required*/
     initproc = <ProcName>;                   /* optional*/
     cleanupProc = <ProcName>;                /* optional*/
     memory = <MemoryOption>;                 /* optional*/
     heap = <HeapType>;                       /* optional*/
     clientdata=<ClientData Option>;          /* optional*/
     flags = <FlagOptions>;                   /* optional*/
};
```

Field descriptions

The fields in a `Library` declaration have the following descriptions:

`id =` *`LibraryIDString`*

This declaration defines the library ID of your shared library. A library ID normally takes the form *`xxyy`*`$Name`, as shown in the code fragment that appears above. It also should include the library's version number to ensure that each library version will have a unique ID. See "TLibraryID" in Chapter 9, "Utility Classes and Member Functions," for more details on the format of a library ID. The `LibraryIDString` parameter is a quoted string, but it may include constants created with the `#define` directive as part of its definition, provided your exports file includes the header files that contain definitions that resolve the constants.

When there are multiple shared libraries with the same library ID, the ASLM uses only one shared library. The others are marked as duplicates and their contents are ignored.

`version =` *`LibraryVersion`*

This declaration contains the version of your shared library. Write the version number in the standard Apple version number form: #.#[.#], followed by either nothing or `[dabf]#` to specify the library's release status—for example, 1.0b2 or 1.1.2d5. The version number may be a constant created with the `#define` directive.

`initproc =` *`ProcName`*

This declaration lets you specify the name of a C function that is called immediately after your shared library is loaded and initialized. The function that is specified in this declaration takes no parameters and returns no value. The function can be in the `A5Init` segment; in this case, the function is unloaded from memory after the library is loaded and initialized.

`cleanupProc =` *`ProcName`*

This declaration lets you specify the name of a C function to be called just before your shared library is unloaded from memory. The function that is specified in this declaration takes no parameters and returns no value. The function must not be in the `A5Init` segment.

`memory =` *`client`*

This declaration specifies that when the C++ `new` operator is used to allocate memory in your shared library, the memory is allocated from the current client's pool. If no `memory` parameter is specified, `memory = client` is the default. For more information on client memory pools, see "Memory Management Classes" in Chapter 8, "ASLM Utility Class Categories."

`memory =` *`local`*

This declaration specifies that any memory-allocation operations carried out by the C++ `new` operator in the library being built will use the library's local pool. For more information on local memory pools, see "Memory Management Classes" in Chapter 8, "ASLM Utility Class Categories."

`heap = default || temp || system || application [,hold][,#]`

This tells the ASLM where you want your library to be loaded into memory. Normally, you should not specify this attribute unless you have a very good reason. However, if your library must run under virtual memory and cannot move in memory (for instance, a network driver), you can specify the `hold` attribute to inform the ASLM that you require the memory where your library is loaded to be "held" under virtual memory. Also, you can optionally specify the size of the heap that you want your library to load into (this option only makes sense for `default` or `temp`). This is useful if you are going to explicitly load and unload library code segments. See "Support for Explicit Segment Loading and Unloading" later in this chapter.

For more information on heap, see "Library Heap Support" later in this chapter.

`clientData =` *`StructureName`* || #

This tells the ASLM that you require per-client static data. You can specify either a number of bytes or the name of a structure. Then whenever you call `GetClientData`, you will be returned a structure of the specified size. The first time the structure is created for a given client, it will be zeroed. After that, you will get back the structure corresponding to your current client. If you specify a structure name, the object file must have the type information available to determine the size of the structure, or an error will be generated. To add type information to the object file, make sure the files are compiled with `-sym on` or `-sym on,nolines`. For more information, see "Per Client Data" in Chapter 7, "ASLM Utilities."

`flags = segUnload || !noSegUnload`

This flag warns that clients may unload segments of your shared library. Normally, the ASLM resolves all jump table references at library load time and removes the jump table from memory. This flag overrides this behavior. For more information, see “Support for Explicit Segment Loading and Unloading” later in this chapter.

`flags = noSegUnload || !segUnload`

This flag specifies that the segments of the shared library will not be unloaded by the client. This is the default setting of the `segUnload` flag. The advantage of this option over `segUnload` is that your library uses less memory when loaded (because the jump table is not needed) and calls that would normally go through the jump table are faster. The disadvantage is that it preloads all library code segments.

`flags = preload`

This flag causes the shared library to be loaded when the ASLM is loaded at boot time. You can also specify `noSegUnload` or `segUnload` when using this flag. For more information, see “Keeping Preloaded Libraries Loaded” later in this chapter.

`flags = loaddeps`

This flag indicates that the ASLM should load all dependent libraries whenever this library is loaded (based on the information in the `'libr'` resource created during the build process). Using this flag guarantees that all libraries on which your library depends, exist and are loaded. It is equivalent to calling `LoadLibraries(false, false)` within your `initproc` except that you are not required to call `UnloadLibraries` to allow your library to unload.

`flags = forcedeps`

This flag acts just like the `loaddeps` flag, but it also forces all of the code segments in the dependent libraries to be loaded into memory. It is equivalent to calling `LoadLibraries(true, false)` within your `initproc`, except that you are not required to call `UnloadLibraries` to allow your library to unload.

`flags = stayloaded`

This flag forces your library to stay loaded. It requires a call to `UnloadLibraries` from within your library to allow your library to unload. It is equivalent to calling `LoadLibraries(doForce, true)` within your `initproc`. The `doForce` parameter is `true` if the `forcedeps` flag is set, otherwise it is `false`.

`flags = system6 || !system7`

This indicates that your library should not be registered if it is installed on a Macintosh running System 7.x. No clients will be able to see any of the classes or function sets in your library. This flag is useful if you have two different versions of your library—one for System 6.x and one for System 7.x.

`flags = system7 || !system6`

This indicates that your library should not be registered if it is installed on a Macintosh running System 6. No clients will be able to see any of the classes or function sets in your library. This flag is useful if you have two different versions of your library—one for System 6 and one for System 7.

`flags = vmOn || !vmOff`

This indicates that your library should not be registered if it is installed on a Macintosh with virtual memory (VM) turned off. No clients will be able to see any of the function sets or classes in your library. This flag is useful if you have two different versions of your library—one to use if virtual memory is on and one to use if virtual memory is off.

`flags = vmOff || !vmOn`

This indicates that your library should not be registered if it is installed on a Macintosh with virtual memory (VM) turned on. No clients will be able to see any of the function sets or classes in your library. This flag is useful if you have two different versions of your library—one to use if virtual memory is on and one to use if virtual memory is off.

`flags = fpuPresent || !fpuNotPresent`

This indicates that your library should not be registered if it is installed on a Macintosh that does not have a floating-point unit (FPU). No clients will be able to see any of the function sets or classes in your library. This flag is useful if you have two different versions of your library—one to use with an FPU and one to use without an FPU.

```
flags = fpuNotPresent || !fpuPresent
```

This indicates that your library should not be registered if it is installed on a Macintosh that has a floating-point unit (FPU). No clients will be able to see any of the classes or function sets in your library. This flag is useful if you have two different versions of your library—one to use with an FPU and one to use without an FPU.

```
flags = mc68000 || mc68020 || mc68030 || mc68040
```

This indicates that your library should only be registered if it is installed on a Macintosh that has the specified processors. You may specify more than one processor. For example, `flags = mc68000, mc68020` will cause your library to be registered only on 68000 or 68020 processors.

```
flags = !mc68000 || !mc68020 || !mc68030 || !mc68040
```

This indicates that your library should not be registered if it is installed on a Macintosh that does not have one of the specified processors. You may specify more than one processor. For example, `flags = !mc68000, !mc68020` will cause your library to be registered only on Macintoshes with a 68030 or higher processor. It is an error to mix not terms (!) with non-not terms—for example, `flags = mc68000,!mc68020`.

`Class` declarations

The `Class` declarations in an exports file identify classes that you want your shared library to export. The following code fragment is an example of a `Class` declaration:

```
Class TLightClass {
      flags = newobject;
};
```

Syntax

The syntax of a full `Class` declaration is:

```
Class <ClassName>
{
     version    = <ClassVersion>;
     flags      = preload, newobject, noVirtualExports,
                  noMethodExports, noExports;
     exports    = <ListOfFunctionNames>;
     dontExport= <ListOfFunctionNames>
     private    = * | <ListOfFunctionNames>
};
```

All fields in the above code fragment except the `<ClassName>` field are optional. Therefore, the smallest possible class declaration has this syntax:

```
Class ClassName;
```

A `#define` must exist for your class's class ID and should be of the form `k`*`ClassName`*`ID`. See "TClassID" in Chapter 9, "Utility Classes and Member Functions," for more details on class ID's. Also, your exports file must `#include` the files that contain the declaration of the C++ class and its parent classes.

Field descriptions

The fields for the class declaration have the following descriptions:

`ClassName`

The name of the class that you want to export.

`version =` *`ClassVersion`*

This declaration defines the version of the class. The version number should have the standard Apple version number form: #.#[.#]. The version number that you use in this field may not include any special release information (such as b2). It can be a constant defined in a `#define` declaration. Also, the version number can be made up of a pair of version numbers separated by either three dots (`...`) or an ellipse (option-;) character. This is called a version range; it is used to specify the lowest version number that the class being defined is backward-compatible with and to specify the current version number of the class. If you do not specify a version number, the version number contained in the class's class ID is used. If the class ID does not specify a version number then it is assumed that the version number of the class is the same as the version number specified in the `Library` declaration. See Appendix D, "Versioning," for more information on versioning and "TClassID" in Chapter 9, "Utility Classes and Member Functions," for more information on class IDs.

`flags = newobject`

This flag specifies that clients are allowed to create an instance of the class with the `NewObject` functions, using the class's class ID. A warning is issued at build time if this flag is set and one of the following is true:

- The class being defined does not have a default constructor.
- The class is abstract (has a pure virtual method).
- The class size cannot be determined from the symbol information in the object file.

`flags = preload`

This flag specifies that an instance of the class should be created whenever the library is loaded. This flag implies the `newobject` flag. A warning is issued at build time if this flag is set and one of the following is true:

- The class being defined does not have a default constructor.
- The class is abstract (has a pure virtual method).
- The class size cannot be determined from the symbol information in the object file.

See “Keeping Preloaded Libraries Loaded” later in this chapter.

`flags = noExports`

This flag specifies that no member functions of this class are to be exported. A client can use this class only with the `NewObject` function if this flag is set and you do not export constructors using the `exports=` option. Also, a client can call only virtual functions in the class, unless you explicitly export methods using the `exports =` option described below.

`flags = noVirtualExports`

This flag specifies that no virtual functions can be exported for this class. This restriction effectively prevents subclasses in separate libraries from explicitly calling inherited functions. It also prevents the calling of virtual functions for stack objects unless the stack object is first dereferenced and cast to a pointer. It does allow normal virtual function calls to be made since they go through the v-table and do not need to be exported. You can explicitly export some virtual functions using the `exports =` option described below.

`flags = noMethodExports`

This flag specifies than no member functions of this class are to be exported, with the exception of the destructor and the constructors. This restriction effectively prevents the class from being subclassed from an application and also prevents subclasses in separate libraries from explicitly calling inherited functions. If also prevents the calling of virtual functions for stack objects unless the stack object is first dereferenced and cast to a pointer.

`exports =` *`ListOfFunctionNames`*

This field contains a comma-separated list of member functions that you want to export from the class being defined. It is normally used to override the `noExports`, `noMethodExports`, or `noVirtualExports` flags for individual methods. All you must specify in this field is the function's name—unless it is a Pascal function, in which case, you must precede the function's name with the keyword `pascal`. The `BuildSharedLibrary` command regards all overloaded variants of a member function as the same function, and therefore exports them all. To export operators, you can use the C++ syntax for specifying operators (for example, `operator+=`). To export constructors, you can use the name of the class. To export destructors, you can use the standard format `~`*`NameOfClass`*.

By default, static methods are not exported. To export static methods, export them in a function set. If you want to export them using the `exports=` option, you must omit the keyword `static`.

`dontExport =` *`ListOfFunctionNames`*

This field contains a comma-separated list of member functions that you do not want to export from the class being defined. All you must specify in this field is the function's name—unless it is a Pascal function, in which case, you must precede the function's name with the keyword `pascal`. The `BuildSharedLibrary` command regards all overloaded variants of a member function as the same function, and therefore will not export any of them. To prevent operators from being exported, use the C++ syntax for specifying operators (for example, `operator+=`). To prevent constructors from being exported, use the name of the class. To prevent destructors from being exported, use the standard format `~`*`NameOfClass`*.

`private = ListOfFunctionNames`

This declares a comma-separated list of member functions that you want to export from the class privately. Any member functions specified in this list are exported, but go into a separate client object file (defined by the `-privateNear` and/or `-privateFar` command-line options to `BuildSharedLibrary`).

```
private = *
```

This declares that all member functions that can be exported should be exported privately. If you have set `noMethodExports`, then all virtual methods are exported privately that are not either explicitly exported publicly by the `exports=` option or that are specifically excluded from being exported by a `dontexport=` option. If you have set `noVirtualExports`, then all non-virtual member functions are exported privately that are not either explicitly exported publicly by the `exports=` option or specifically excluded from being exported by a `dontexport=` option. If you have neither flag set, than all member functions of the class are exported privately that are not either explicitly exported publicly by the `exports=` option or specifically excluded from being exported by a `dontexport=` option. It is an error to use this switch if the `noExports` flag is set.

`FunctionSet` declarations

To export functions from your shared library, use `FunctionSet` declarations in the exports file. The following code fragment is an example of a `FunctionSet` declaration:

```
#define kLightFunctionSet "appl$TrafficLightFSet,1.1"
FunctionSet LightFSet {
      id =        kLightFunctionSet;
      exports =   NewTrafficLight,
                  FreeTrafficLight,
                  GetLight,
                  SetLight,
                  DrawLight,
                  AdjustTrafficLightMenus,
                  DoTrafficLightMenuCommand;
};
```

The following code fragment is an example of a function set for functions written in Pascal:

```
#define kLightFunctionSet "appl$TrafficLightFSet,1.1"
FunctionSet LightFSet {
      id =        kLightFunctionSet;
      exports =   pascal NewTrafficLight,
                  pascal FreeTrafficLight,
                  pascal GetLight,
                  pascal SetLight,
                  pascal DrawLight,
                  pascal AdjustTrafficLightMenus,
                  pascal DoTrafficLightMenuCommand;
};
```

Syntax

The syntax of a full function set declaration is:

```
FunctionSet <FunctionSetName>
{
     id =            <FunctionSetID>;             /* required*/
     interfaceID =   <InterfaceIDString>;         /* optional*/
     version =       <FunctionSetVersion>;        /* optional*/
     exports =       <ListOfFunctionNames>;       /* optional*/
     dontexport =    <ListOfFunctionNames>;       /* optional*/
     private =       * | <ListOfFunctionNames>;   /* optional*/
};
```

Field descriptions

The fields in a function set declaration have the following descriptions:

`FunctionSet` *`FunctionSetName`*

This field provides a unique name for your function set during the linking process. This name is used in the client object file for module names generated by `BuildSharedLibrary`. If the same *`FunctionSetName`* is used by more than one function set, the client will only be able to link with one of the function set’s client object files. For this reason you should choose a unique name.

`id =` *`FunctionSetID`*

This declaration defines the ID of the function set. A function set ID normally takes the form *`xxxx:yyyy`*`$SomeName`. It should also include the function set's version number. For more details on the format of a function set ID, see "TFunctionSetID" in Chapter 9, "Utility Classes and Member Functions." This ID string is a quoted string, but it may include constants created with the `#define` directive as part of its definition, provided you include the header files containing the definitions that resolve the constants. If you do not include an `id =` declaration in your function set declaration, a search is made in included header files for constants (created with the `#define` directive) with a name that matches `k`*`functionSetName`*`ID`. If such a name is found, it is assumed to be the function set ID for the function set. If the function set ID cannot be determined, an error occurs at build time.

`interfaceID =` *`InterfaceIDString`*

This declaration establishes an interface for your function set. The format of `InterfaceIDString` is the same as `FunctionSetID`. Normally, you use this to specify which function sets have the same interface. You can then use `GetFunctionSetInfo` to find all of the function sets with the same interface. Combined with the `GetFunctionPointer` and `GetIndexedFunctionPointer` functions, this allows you to choose which function to call from among function sets with the same interface. For more details, see "Getting Information about Function Sets" and "Calling Functions by Name" in Chapter 7, "ASLM Utilities."

`version =` *`FunctionSetVersion`*

This declaration defines the version of the function set. The version number should have the standard Apple version number form: #.#[.#]. The version number that you use in this field may not include any special release information (such as b2). It can be a constant defined in `#define` declaration. Also, the version number can be made up of a pair of version numbers separated by either three dots (...) or an ellipse (option-;) character. This is called a version range. It is used to specify the lowest version number with which the function set is backward-compatible and to specify the current version number of the function set. If you do not specify a version number, the version number contained in the function set's function set ID is used. If the function set ID also does not specify a version number, it is assumed that the version number of the function set is the same as the version number specified in the `Library` declaration. See Appendix D "Versioning" for more information on version numbers, and "TFunctionSetID" in Chapter 9, "Utility Classes and Member Functions," for more details on the format of a function set ID.

`exports = `*`ListOfFunctionNames`*

This field declares a comma-separated list of functions that you want to export in this function set. All you must specify in this field is the function's name—unless it is a Pascal function, in which case, you must precede the function's name with the keyword `pascal`. If this field is omitted, all functions in the `InputObjectFile` are exported automatically.

You may export a function by name by using the keyword `external` in front of the function name. This allows the function to be used by `GetFunctionPointer`. However, you may not export C++ class member functions by name.

If you are exporting C++ functions, `BuildSharedLibrary` regards all variants of an overloaded member function as the same function, and therefore exports them all (unless you use the `-c` option on the `BuildSharedLibrary` command line).

If you want to export C++ class member functions in a function set, you should precede the name of the member function with the name of the class, using the format *`ClassName`*`::`. The `-c BuildSharedLibrary` option is ignored when exporting member functions and all overloaded variants of the member function are exported. To export C++ operator overloads, use the standard C++ syntax (for example, `operator+=`). To export constructors, use *`ClassName`*`::`*`ClassName`*. To export destructors, use the standard form *`ClassName`*`::~`*`ClassName`*.

If you want to export all member functions of a C++ class, use `class` *`ClassName`*. If you want to export all static member functions of a C++ class, use `static` *`ClassName`*.

Exporting a class's member functions through a function set can be useful when the class has no constructor or destructor, or when the constructor or destructor of the class is inline. In these cases, classes cannot be exported in the normal way.

When exporting a static method of a class or a static function, omit the keyword `static`.

You cannot export C++ global operators in a function set. You can export C++ cast operators, but only if they are predefined. Cast operators that are not predefined are not allowed.

`dontexport =` *`ListOfFunctionNames`*

This declares a comma-separated list of functions that you do not want to export in this function set. It has the same syntax as the `exports=` option, except that the `static`, `class,` and `extern` keywords are not valid.

This field is useful if you have omitted the `exports=` option (which causes all functions in the `InputObjectFile` to be exported) and you want to prevent certain functions from being exported.

`private =` *`ListOfFunctionNames`*

This declares a comma-separated list of methods that you want to export from the function set privately. Any methods specified in this list are exported, but go into a separate client object file (defined by the `-privateNear` and/or `-privateFar` command-line options to `BuildSharedLibrary`). If you have not defined an `exports=` or `dontExport=` clause, then all other functions are exported publicly.

`private = *`

This declares that all functions that can be exported should be exported privately. If you have not defined an `exports=` or `dontExport=` option, then all of the functions are exported privately. If you have an `exports=` option, then the functions declared there are exported publicly, and all others are exported privately. If you have a `dontExport=` option, then the functions declared there are not exported at all, and all others are exported privately. If you have both options, those in the `dontExport=` option are not exported, those in the `exports=` option are exported publicly, and all others are exported privately.

Library environment flags

When you declare a library in the exports file, you can use the following flags to define the environment that must exist for the ASLM to register the library and its function sets and classes: `vmOn`, `vmOff`, `System6`, `System7`, `FPUpresent`, `FPUNotPresent`, `MC68000`, `MC68020`, `MC68030`, and `MC68040`. For example, the `vmOn` flag means that virtual memory must be turned on, and `System6` means that System 6 must be running. By using these flags, you can create versions of a library that can be used in different situations (such as one version for System 6 and another for System 7).

The flags are broken up into four groups: the virtual memory group (`vmOn` and `vmOff`), the system group (`System6` and `System7`), the floating-point unit group (`FPUpresent` and `FPUNotPresent`), and the processor group (`MC68000`, `MC68020`, `MC68030`, and `MC68040`). If one or more flags from the same group are specified, the library can be used only when the condition specified by one of the mentioned flags exists. For example, if you only specify `MC68020` then your library will only run under the 68020 processor and no others. If you also want it to run under the 68030 then you should also specify the `MC68030` flag.

You can also specify that a library is not to be used in a particular environment by using the construct !*flagname*. For example, a `!MC68000` flag means the library can run on anything but an MC68000.

The individual flags are described in “Library Declaration” earlier in this chapter.

Putting multiple libraries in a library file

Each shared library in a shared library file contains three types of resources: a `'libr'` resource, a `'libi'` resource, and a set of three or more code resources. The `'libr'` resource describes the classes and function sets in the library. The `'libi'` resource describes the library’s dependencies on other libraries. Code resources contain the implementation of the library. Although a shared library file can contain more than one shared library, each shared library has its own `'libr'` and `'libi'` resources and its own set of code resources.

Usually you use the MPW `Rez` command to create a library file that contains multiple libraries. You must include each `'libr'` resource, giving each `'libr'` and `'libi'` resource a unique ID if there is more than one. You will also need to give each code resource type a unique type. The resource ID and code resource type must be specified when building the library. They are `BuildSharedLibrary` options. Do not change the resource ID of the `'libr'` or `'libi'` resource when using `Rez` to create your shared library file. Also, do not change the resource type of the code resources.

The LibraryManager.o file

The LibraryManager.o file illustrated in Figure 4-1, "Building a Client," is an MPW library file supplied for ASLM client and library developers. It contains

- client object file (.cl.o) code for shared libraries supplied with the ASLM
- routines defined in the ASLM header files
- the `DynamicCodeEntry` routine, which performs certain initializations and must be linked with and be the entry point for shared libraries
- other behind-the-scenes routines that are used internally

The LibraryManager.o file should be linked before any C libraries are linked. It should also be linked before CPlusLib.o unless you want to use the global `new` operator supplied by CPlusLib.o. See "Using the ASLM Global `new` and `delete` Operators" in Chapter 6, "Using the ASLM," for more details.

The LibraryManager.n.o file is similar to the LibraryManager.o file, except that it is meant only for model near clients and, therefore, is not compiled with model far. Since shared libraries must always be compiled with model far, they will never link with LibraryManager.n.o.

LibraryManager.debug.o and LibraryManager.debug.n.o

LibraryManager.debug.o and LibraryManager.debug.n.o are debug versions of the library files and contain debugger breaks and MacsBug symbols useful when trying to debug clients and shared libraries.

Library heap support

The following table shows that the time of the load and the `heap=` option that was used in the `Library` declaration determines the heap into which a shared library will load. The top row of the table specifies the possible load times. The leftmost column specifies the `heap=` option that was used.

	Preload time	**INIT time**	**Single finder**	**System 6**	**System 7**
default	System	System	Application	temp	temp
temp	System	System	System	temp	temp
system	System	System	System	System	System
application	System	Application	Application	Application	Application

The following load times are possible (top row):

- Preload Time is when the ASLM is loading at boot time and is preloading libraries. In other words, the library is loading because it set its preload flag, or because another preloaded library caused it to load.
- INIT time means the library is loading because an INIT is using it (directly or indirectly).
- Single Finder is System 6 with MultiFinder turned off.
- System 6 is System 6 with MultiFinder turned on.
- System 7 is System 7.

Single Finder, System 6, and System 7 load times are all after the system has finished booting. In other words Preload Time and INIT Time take precedence over them.

The following `heap=` options are possible:

- “System” is the System heap.
- “temp” is a subheap of the MultiFinder (Process Manager) heap.
- “Application” is the application heap.

Do not set a library to load into the system heap unless you know that it will only be loaded when the system heap can grow or when there is enough memory reserved for the library. The System heap does not grow during INIT Time, or while running under System 6 (including Single Finder). It will grow during preload time and under System 7.

Temp heaps are similar to application heaps in the way they are allocated and where they exist in memory. They are somewhat misnamed because there is nothing temporary about them. They are called temp heaps because they are allocated using MultiFinder (also called the Process Manager) temporary memory.

If you are debugging using MacsBug and your shared library is not loaded in the system or application heap, it can sometimes be difficult to locate the MacsBug symbols for your shared library. The best way to locate them is to use the MacsBug `hx` command to switch to the MultiFinder heap so that you can see all the symbols for any shared library loaded in temp memory. You will also be able to see all the symbols for all currently running applications, since they too are in subheaps of the MultiFinder heap. The MultiFinder heap is always located immediately after the system heap in memory. The best way to find it is to use the MacsBug `hz` command to list all the heap zones, find the system heap in the list (it should be first), and then add 1 to the value specified as the end of the system heap. This is the value you want to pass to the `hx` command to switch to the MultiFinder heap.

For more information on the `heap=` option, see “Library Declaration” earlier in this chapter.

Log file support

Since exporting more functions, adding constructors to classes, adding more non-virtual functions to classes, modifying or moving virtual functions in classes, or changing the size of a class can cause incompatible libraries to be built, a logging mechanism has been built into the build procedure for a library. This allows the new library to be built in a backward-compatible manner to the previous version of the library, if at all possible. There are three switches to the `BuildSharedLibrary` script to control logging.

```
-logout <OutputLogFileName>
-log <InputLogFileName>
-dolog
```

logout

The `logout` switch specifies the output log file. The output log file is an ASCII text file that shows where various functions, v-tables, and so on, are being exported.

log

The `log` switch specifies an input log file. The log file is used to control the generation of the new library.

dolog

The `dolog` switch actually enables the logging operations. (This is so that you can specify `-logout` or `-log` in your makefile, but nothing is done until you alias `BuildSharedLibrary` to be `BuildSharedLibrary -dolog`, or something similar.)

Your output library is built so that it is compatible with the version of the library which created the input log file. Warnings tell you of any incompatibilities between old and new libraries, as well as any versioning problems. However, the build will never be aborted due to these warnings. It is your library, and you may want to make nonconforming version numbers known.

Speeding up builds

The `BuildSharedLibrary` script only rebuilds the entire library if it notices that the library's input object file or exports file has changed. Otherwise, `BuildSharedLibrary` merely links the shared library. Not rebuilding the entire library is useful when an object file that must be linked with a library has changed. In such a case, only relinking is needed. You need to specify the `-obj` parameter if you want to use this feature.

If you are building a library in two steps—that is by executing both `BuildSharedLibrary` and `LinkSharedLibrary`—this strategy yields no benefit, since you do not have to call `BuildSharedLibrary` unless the input object file or the exports file has changed.

If the library's exports file or input object file does change, `BuildSharedLibrary` builds the entire library; that is, `BuildSharedLibrary` processes the exports file and input object file and creates new client object files and intermediate files. (For an explanation of intermediate files, see the `-obj` option in "Using BuildSharedLibrary" earlier in this chapter.

Using the `-keepClientFiles` option

The `BuildSharedLibrary` command always creates new client object files. However, if you use the `-keepClientFiles` option, it discards the object files if they have the same contents as the existing ones. This procedure does not really speed up the build of your library, but it does let you rebuild your shared library without changing the modification dates of the client object files. This means you do not have to relink clients that are dependent upon your client object files.

Using `-keepClientFiles` is only useful if the clients do not share a makefile with the library. Otherwise the clients will still be relinked even if the client object file does not change. This is because the client is dependent on the client object file, the client object file is dependent on the library, and the library is dependent on the input object file and the exports file. Thus if both the client and the library are in the same makefile, the client object file appears out of date to the `Make` command whenever the exports file or input object file changes. This results in the client being rebuilt even if `BuildSharedLibrary` did not change the modification date of the client object file.

Splitting up the makefile solves this problem because the client's makefile will not know what the client object file depends on. So the client's makefile relinks the client only if the client object file changes. This is generally worth the effort only if you have a considerable number of clients that depend on the library, or if it takes the client a long time to link. The TestTool and Inspector programs that are provided on the *ASLM Examples* disk provide examples of how to use `-keepClientFiles` and write the makefile in this manner.

Linking with model near code

You must be careful when linking model near object files with shared libraries. This can be a problem when you link with certain libraries supplied by MPW, since MPW libraries are compiled using model near. Normally, a shared library is not in its global world when it is called (that is, the A5 world is not set up correctly for calls to the routines in the model near MPW library to succeed). This means that the shared library must enter its global world before it calls any model near code that contains references to global variables or any model near code that makes a call to code in another code segment (an intersegment call).

You can avoid the intersegment call problem by merging all your shared library's implementation code segments into one code segment using the linker's `-sg` option. But you still must enter the library's global world before you call model near code that has references to any global variables.

To enter the library's global world so that you can call a function that is compiled using model near, call `OpenGlobalWorld` before you call the model near function, and call `CloseGlobalWorld` after the model near function returns.

```
GlobalWorld saveworld;
saveworld = OpenGlobalWorld();
     /* make model near call here  */
CloseGlobalWorld(saveworld);
```

If you choose not to merge your implementation code segments into one code segment, you must use the `flags=segUnload` option when you declare your library in the library's exports file. See "Writing an .exp File" earlier in this chapter for an explanation of the `flags=segUnload` option.

Using MPW libraries

Shared libraries often run into problems when calling standard C library functions—for example, `sprintf`, `sscanf`, `malloc`, `atan2`, and other functions which require linking to the StdCLib.o library and other MPW libraries. There are a few problems with using these routines:

- They are not compiled using model far.
- Some of them make callbacks into MPW.
- Some of them allocate memory and never free it.
- Some of them use globals.

The problems with linking with model near code are explained above in “Linking With Model Near Code.” `BuildSharedLibrary` and `LinkSharedLibrary` take care of the jump table problem by forcing all of the MPW libraries to be merged into the Main code segment.

The problem with the MPW callbacks is that when they are called from a shared library, the environment is not set up for them to work. Routines that use MPW callbacks include any of the i/o routines such as `fprintf` when they are used with `stdout`, or `stderr` unless they have been redirected to a file. This includes the routines that use one of these by default, such as `printf`. You might want to try using `Trace` instead to display the output in the TraceMonitor’s Trace window. Another solution is used by the ASLM’s TestTool example. It sets a print function for each object that it creates. This print function exists in the MPW tool and simply sends the output to `stdout`. This allows the object to essentially do a `printf`. This print function is called `myPrintFunc` and can be found in TestTool.cp.

The problem of some of the routines allocating memory that does not get freed is one of the more annoying ones. Some of the routines cause some memory to be allocated the first time one of the routines in a “family” is called. A pointer to this memory is stored in a global so it can be reused on successive calls. The libraries rely on the fact that when the application quits, the memory automatically gets freed up when the application heap is freed. If you call one of these routines from a shared library or any stand-alone code resource, the memory gets allocated from the application heap and is not freed up until the heap that it was allocated from goes away (usually when the application quits).

One of the memory allocation offenders is `sprintf` and others in its family (`scanf`, `fprintf`, `sprintf`, and so on). They all share a buffer that gets allocated the first time one of them is called. Another offender is `malloc`, which creates a big chunk of memory from which to allocate little chunks.

Libraries that are shared could crash if they allocated the memory from one application heap and then, while a second application is also using the library, the first application quits. Now the pointer is invalid but the library is not aware of this. Libraries that are only used by one application at a time will show no memory leak once the application quits, so they do not need to worry about this problem unless the application causes the library to repeatedly load and unload.

Currently, there is no general solution to this problem. You can get around the `sprintf` problem by using the ASLM `sprintf` routine. You can get around the problem with `malloc` by using memory pools.

Segmentation and run-time architecture

Shared library classes are compiled and linked using model far and are linked as multiple code resources, with a jump table for dispatching between code resources similar to the Macintosh application segmentation model. In the resource file, the jump table corresponds to the `'CODE'` 0 segment of an application. In the case of a shared library, the resource type is usually `'code'` (spelled in lowercase letters), but that is up to the developer. However, the resource type should never be `'CODE'` (spelled in uppercase letters); that may result in accidentally launching the library as an application.

A shared library always has at least two code segments besides the jump table: one that contains initialization code and one that contains the implementation. A shared library can have as many code segments as you wish; however, unless you plan to explicitly load and unload your library's code segments, it is generally best to have only one implementation segment. See "Support for Explicit Segment Loading and Unloading" later in this chapter for more details.

Figure 5-2 shows the segmentation of a shared library.

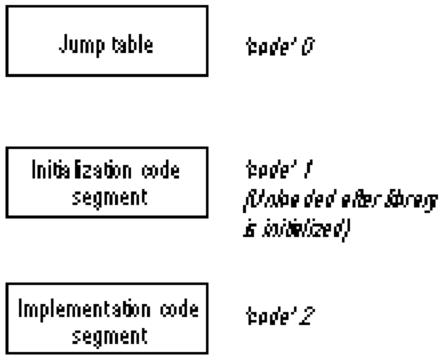

Jump table
'code' 0
Initialization code segment
'code' 1
(Unloaded after library is initialized)
Implementation code segment
'code' 2

Figure 5-2 Code segments of a shared library

Support for explicit segment loading and unloading

As Figure 5-2 illustrates, every shared library has at least three code segments: a jump table (segment 0), an initialization segment (segment 1), and an implementation segment (segment 2). However, a library can break its implementation segment into more than one code segment so that its entire implementation does not have to be in memory at the same time.

For example, when you design a shared library, you might put all code that handles a certain task (for instance, printing) into a separate code segment. You could call that segment code Segment 3. Then a call to any code in Segment 3 automatically causes that segment to be loaded. Once a task in a numbered code segment is completed, you can unload the segment by calling `UnloadSegmentByNumber`.

Generally, a better method for unloading unneeded code is to put the code that handles the task in a separate library and to encapsulate it with a C++ class. Then the code is loaded automatically when you instantiate its class and is unloaded when you delete the class. If you do not want to use C++ or you do not feel that a task is big enough to warrant its own library (but it is big enough to put in a separate code segment and unload when it is not in use), then using segmentation as described above is an acceptable alternative.

Note: When running under MultiFinder, by default a shared library is loaded into a heap that is a subheap of the MultiFinder heap. One heap is created for each shared library and each library's heap is large enough to hold all code segments of the library. This means that in the default case, explicitly unloading code segments does not free up memory that can be used for other purposes. In order to make unloading library code segments worth while, you either need to specify that the library's code load into the system or application heap, or you need to specify the size that the library's heap should be. Both of these tasks are accomplished by using the `heap=` option described in "Library Declaration" earlier in this chapter.

IMPORTANT If you call code in a segment that is not currently loaded and there is not enough memory to load the segment, or the segment load occurred at non-System Task time, an exception is raised. For this reason, you should always have an exception handler installed before attempting to call unloaded code. It is up to the library writer to decide if the library or its clients should be in charge of installing the exception handler. To avoid needing the exception handler, you should call one of the `LoadCodeSegmentXX` routines. This ensures that the code is loaded before you call the code. See "Exception Handling" in Chapter 7.

Library code segments can be explicitly loaded and unloaded by using the following functions:

```
OSErr  LoadCodeSegmentByNumber(TLibrary*, short segmentNumber);
OSErr  LoadCodeSegmentByName(TLibrary*, ProcPtr theRoutine);
OSErr  UnloadCodeSegmentByNumber(TLibrary*, short segmentNumber);
OSErr  UnloadCodeSegmentByName(TLibrary*, ProcPtr theRoutine);
```

All four of these functions take a pointer to a library's `TLibrary` object. See "Getting a Library's TLibrary's Object" in Chapter 7, "ASLM Utilities," for details on how to get a library's `TLibrary` object.

The `LoadSegmentByNumber` function takes a `segmentNumber` parameter that specifies the segment to load. The `LoadSegmentByName` function takes a `ProcPtr` parameter that holds the address of a function in the segment to be loaded. Both `LoadSegmentByNumber` and `LoadSegmentByName` return an `OSErr` data type. If the segment cannot be loaded, a `kCouldNotLoadCode` or `kOutOfMemory` error is returned. If `kCouldNotLoadCode` is returned, the specified code segment number is invalid; `kOutOfMemory` means that there was not enough memory available to load the code.

The `UnloadSegmentByNumber` function takes the segment number to unload as a parameter. It returns `kCodeNotLoaded` if the segment number is invalid or if the code segment is already loaded.

The `UnloadSegmentByName` function takes a `ProcPtr` parameter that holds a pointer to the jump table entry of a function in the segment that you want to unload. If the specified address is not in a loaded segment, `UnloadSegmentByName` returns a `kCodeNotLoaded` value.

IMPORTANT The `ProcPtr` parameter of the `UnloadSegmentByName` function is a pointer to the jump table entry for a function, not the address of the function itself. So you must obtain the address of the function from code that lies outside the code segment of the routine whose address you want to obtain. In other words, you must make an intersegment reference to the routine, not an intrasegment reference. This generally means that you should not try to unload the code segment from within the code segment. The `LoadSegmentByName` function has this same restriction, but this is seldom a problem because you do not normally attempt to load a code segment while code within that segment is being executed. However, if you have just merged two code segments into one segment, you may find that this is what you are trying to do.

Keeping preloaded libraries loaded

If a shared library's `preload` flag is set, the ASLM loads the library at boot time. However, unless you take special steps to keep the library loaded, it unloads immediately afterwards. An easy way to keep the library loaded is to call `LoadLibraries` from the library's `initproc`, making sure that the `doSelf` parameter in `LoadLibraries` is set to `true`. Then your library will not unload until you call `UnloadLibraries`.

An easier way to keep a library loaded is to also set the library's `stayLoaded` flag. Setting a library's `stayLoaded` flag has the same effect as executing the following call from an `initproc`:

```
LoadLibraries(false, true)
```

If you also set the library's `forcedeps` flag then it is the same as executing the call:

```
LoadLibraries(true,true)
```

Another way to keep your shared library loaded is to call `LoadClass` or `LoadFunctionSet` from the library's `initproc` on a function set or class implemented in the shared library. The shared library will stay loaded until you call `UnloadFunctionSet` or `UnloadClass`.

Libraries with the `preload` flag set are preloaded only at boot time. If the ASLM is loaded at any subsequent time, a library with the `preload` flag set is not preloaded.

Having a preloaded class in your library is enough to keep the library loaded until the instance of the class is deleted. You can also get the same result by creating an object implemented in the shared library from within the library's `initproc`.

Library global variables

The ASLM allocates global variables for libraries from the global pointer down. The jump table is above the global pointer (on the Macintosh, the A5 register is used for the global pointer). Figure 5-3 shows the ASLM's global world. This is the same as the Macintosh application model for the global world.

The global world's memory is allocated and initialized automatically when the library is loaded.

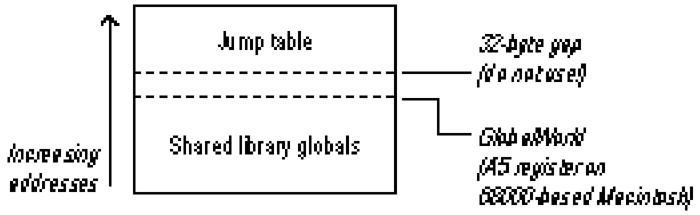


Figure 5-3 A shared library's global world

Using static objects in shared libraries

Static objects in shared libraries can be either shared or unshared classes. (Shared classes are classes that the ASLM knows about because they are exported by a shared library.) Static objects in a shared library are automatically constructed when the library is loaded and are automatically destroyed when the library is unloaded. Static objects that are shared classes are not permitted outside shared libraries, such as in INITs and applications.

WARNING Do not attempt to merge the "Static_Constructors," "Static_Destructors," or "%_Static_Constructor_Destructor_Pointers" code segments into any other code segment in your shared library. These code segments are all created automatically by C++ when your shared library uses static objects.

Registering shared library files

When you have written and built a shared library file, you can make it accessible to clients in one of four ways:

- If you are running System 7, you can place the shared library file in the Extensions folder.
- If you are running System 6, you can place the shared library file in the System Folder.
- You can place the shared library file in any folder that has been registered as a shared library folder.
- You can register a file as a shared library file. In this case the shared library file can be located in any folder.

For information on registering files and folders, see “Registering Shared Library Files and Folders” in Chapter 7, “ASLM Utilities.”

The easiest way to register a shared library is to place it in the System 7 Extensions folder or the System 6 System Folder. Then, at run time, when a client calls a function implemented in the shared library, the ASLM can find the function that was called and the function is executed.

6 Using the ASLM

This chapter provides details on certain runtime related topics that were not appropriate for other chapters. These topics include:

- loading shared libraries
- using the ASLM under System 6 and 7
- using shared libraries
- creating objects
- the TDynamic family of base classes
- using global `new` and `delete` operators
- virtual functions

Loading shared libraries

Shared libraries are loaded on demand:

- When an C++ object implemented by a shared library is created.
- When a shared library is loaded and it implements a class whose parent class is in another shared library, then the parent class's shared library is loaded.
- When a function in a function set is called.
- When the library is explicitly loaded by `LoadClass`, `LoadFunctionSet`, or `LoadLibraries` as described in "Loading and Unloading Shared Libraries" in Chapter 7, "ASLM Utilities."
- At boot time, if the shared library's `preload` flag is set. (If a shared library is loaded at boot time because the library's `preload` flag is set, you must take steps to ensure that the library is not unloaded immediately afterwards. For more information, see "Keeping Preloaded Libraries Loaded" in Chapter 5, "Writing and Building Shared Libraries.")

When a shared library is loaded, the ASLM initializes the shared library. Initialization includes: calling the `DynamicCodeEntry` function supplied in the LibraryManager.o file, allocating storage for library global variables, initializing library global variables, initializing the library's jump table, and calling any static initializers for static objects that the library may have. The code segments that implement the shared library may not actually be loaded depending on how the shared library was built and why it was loaded. However, the code segments will be loaded when the code within them is actually called. (For more information on code segment loading see "Support for Explicit Segment Loading and Unloading" in Chapter 5, "Writing and Building Shared Libraries.")

When a shared library is no longer being used, the ASLM unloads the code from memory automatically. If the shared library is subsequently needed again, it is reloaded and relinked automatically.

The ASLM keeps track of use counts for all exported classes so it can tell if all instances of a class have been deleted and the class is no longer is use. However, when function sets are used, they are considered to be in use until the client calls the `CleanupLibraryManager` or `ResetFunctionSet` function. The `CleanupLibraryManager` call is described in "Creating and Deleting the Local Library Manager" in Chapter 7, "ASLM Utilities." The `ResetFunctionSet` call is described in "Loading and Unloading Shared Libraries" in Chapter 7, "ASLM Utilities."

The ASLM does not immediately unload unused libraries. On the Macintosh, the ASLM attempts to unload libraries once each second at System Task time.

Using the ASLM under System 6 and System 7

The ASLM supports system software versions 6.0.5 through 6.0.8, as well as system software versions 7.0 and 7.1.

Under System 6, the ASLM works with the Finder as well as with MultiFinder. When running under Finder, any libraries that an application causes to load are loaded into the application's heap and are forced to unload when the application quits, even if the application leaves some objects undeleted.

Under System 6, the `EnterSystemMode` call does not prevent any files that you have explicitly opened from being closed when the application that was running when you opened the file quits. However, library files that are opened by calling `Preflight` or `OpenLibraryFile` remain open when the application that was running when you opened the file quits. (For more information on `Preflight` and `OpenLibraryFile`, see "Library File and Resource Management" in Chapter 7, "ASLM Utilities." For more information on `EnterSystemMode`, see "Entering and Leaving System Mode" in Chapter 7, "ASLM Utilities.")

Using shared libraries overview

The ASLM allows clients to use function sets and classes implemented in shared libraries.

Shared libraries can export C++ classes that C++ programs can dynamically link with. Clients written in non–object-oriented languages can also use the C++ class as long as the developer of the shared library provides a procedural interface to the classes.

Shared libraries that are intended to be used by clients written in non–object-oriented languages can export dynamically linkable procedures and functions by using function sets. Non–object-oriented programs can share function set implementations in the same ways that object-oriented programs share classes.

Before a client can use the functions or classes that are implemented in a shared library, the client must do the following:

- Include the header file that defines the functions and classes that the shared library contains.
- Link statically with a client object file that contains the stubs that are responsible for handling the dynamic linking of functions.

- Make sure that the shared library is registered or is in a folder registered with the ASLM at run time.
- Register itself as an ASLM client by calling the ASLM function `InitLibraryManager`.

When all of the above conditions are fulfilled, a client can create objects and call functions implemented in shared libraries. The client can create objects implemented in shared libraries by using the `new` operator (described in “Creating Objects” later in this chapter) or by using automatic variables (that is, stack variables). Alternatively, the client can create objects by calling the `NewObject` function. The `NewObject` function creates objects by class ID. When you create an object with `NewObject`, you do not need to link with the client object file. See “Creating an Object Using NewObject” later in this chapter and “Using NewObject″ in Chapter 7, “ASLM Utilities,” for more information on the `NewObject` function.

When a client creates an object or calls a function that is implemented in a shared library, the ASLM checks to see if the shared library that implements the desired object or function is loaded. If the shared library is not loaded, the ASLM loads it. The loading of shared libraries is transparent to the client.

When a shared library is no longer being used and all clients using the library have deleted all instances of classes that are implemented in the library, the ASLM unloads the code from memory automatically. If the shared library is subsequently needed again, it is reloaded and relinked automatically. See “Loading Shared Libraries” above for more information on when shared libraries are loaded and unloaded.

Sometimes a shared library may fail to load, either because the implementation cannot be located or because there is not enough memory for the shared library. If a shared library fails to load when a function in a function set is called or when an instance of a class is created, the ASLM will raise an exception that the client must catch. The default exception handler that `InitLibraryManager` installs detects this condition and forces the application to quit. A client can prevent this behavior by installing its own exception handler or by preloading needed libraries. For more information about exception handlers, see “Exception Handling” in Chapter 7, “ASLM Utilities.”

Creating objects

A client can create an instance of a shared class dynamically by using the `new` operator. A client can also allocate the object on the stack—that is, as an automatic variable. Static instances of shared classes are also allowed, but only within a shared library.

Creating an object with the `new` operator

When you create an instance of a shared class, you will normally use the ASLM global `new` operator. You can use the ASLM global `new` operator with or without specifying a memory pool. If you do not specify a pool, the ASLM uses the default pool. (For more information on memory pools, see Chapter 8, “ASLM Utility Class Categories.”)

All instances of classes that inherit from a class in the `TDynamic` family are allocated with the `TDynamic` class’s `new` operator. All other objects are allocated using the standard C++ library `new` operator, unless the GlobalNew.h header file is included, in which case the ASLM global `new` operator is used instead.

The `TDynamic new` operator is the same as the ASLM global `new` operator that is declared in GlobalNew.h. It allocates memory from a memory pool. If a pool is specified with the `new` operator, then that pool is used. Otherwise the default pool is used.

It is highly recommended that all C++ shared libraries `#include` GlobalNew.h so all memory allocation is done out of pools. Otherwise the C++ memory allocator is used, and it can cause problems when used from a shared library. For more information, see “Using the ASLM Global `new` and `delete` Operators” later in this chapter.

When you create an object with the ASLM global `new` operator, you can specify the memory pool from which you want to allocate the object, or you can simply let the ASLM use the default pool. For example:

```
TMyClass* myObject = new (myPool) TMyFirstClass;  // from myPool
TMyClass* myObject = new TMyClass;                // from default pool
```

Creating an object using `NewObject`

You can call the `NewObject` function to create an object even if you do not know the class of the object at compile time. The `NewObject` method takes a class ID string as a parameter; the content of the string can be determined at run time. (A class ID is a string that identifies the class to create.)

You can create an object using `NewObject` only if the `newobject` flag is set for the class. The `newobject` flag is set on a per-class basis when a shared library is built. (The `newobject` flag is described in “Class Declarations” in Chapter 5, “Writing and Building Shared Libraries.”) For an object of a given class to be created using `newobject`, the class must have a constructor with an empty argument list.

For classes that require parameters to be passed to the constructor, the class can also provide an initialization method that the `NewObject` caller must call after creating the object.

The following is an example of `NewObject`:

```
object = (TBaseClass*) NewObject(ClassID("esd:sample$TMyFirstClass"));
```

As another example, a client can use `theClassID` as a parameter pointing to a string like the one in the previous example:

```
object = (TBaseClass*) myLibManager-> NewObject(ClassID(theClassID));
```

The `NewObject` function is described in more detail in “Using NewObject” in Chapter 7, “ASLM Utilities.”

Creating stack objects

You can create objects on the stack just as you normally do in C++: by declaring class variables in your routines. You can also create objects that are fields of another object. For example:

```
foo()
{
  TMyClass x;
  x.DoThisAndThat();
}
```

If you use the `STACKOBJECTONLY` macro in a class declaration, the macro informs the compiler that instances of the class will be used only as stack objects. This will make the class’s constructor and destructor much smaller since they do not have to be concerned with allocating or freeing memory.

An example:

```
class TMyClass
     {
     STACKOBJECTONLY

     public:
                              TMyClass ();
          virtual             ~TMyClass ();
          virtual short       Hello() const;
     };
```

When you use stack objects, virtual function calls will not go through the v-table. Instead, the implementation of the virtual function will be called directly, since C++ knows the class type (polymorphism does not take place) and exactly which member function to call. This requires that C++ clients statically link with the implementation of the virtual function just as they do with non-virtual functions. This is one reason you have the option of exporting virtual function stubs when creating a shared library. The client using the stack object will statically link with the virtual function stub so the virtual function call will be made in a way similar to a function set call (this is how non-virtual functions are called).

You can fool C++ into using the v-table for stack object virtual function calls by dereferencing the stack object to make it a pointer. For example: `(&myStackObject)—>DoSomething`. This is much more efficient than making the call through a virtual function stub.

WARNING You must not create objects on the stack in the same routine that calls `InitLibraryManager` unless the stack object is declared after the call to `InitLibraryManager`. Also, you must not create stack objects in the same routine that calls `CleanupLibraryManager` unless the stack object is declared in a nested block that appears before the call to `CleanupLibraryManager`.

Creating static objects

You cannot create static objects of shared classes outside a shared library. This is because static objects are created when the global world is created—and this always takes place before `InitLibraryManager` is called. However, static objects are allowed in shared libraries for both shared and unshared classes. They are automatically constructed when the library is loaded and are automatically destroyed when the library is unloaded.

Creating an object by setting a class's preload flag

If you want an instance of a class to be created automatically when a shared library is loaded, you can set the `preload` flag for the class. If the `preload` flag for a specific class in a shared library is set, an instance of the class is created immediately after the library is loaded. (The `preload` flag is covered in "Class Declarations" in Chapter 5, "Writing and Building Shared Libraries.")

The `TDynamic` family of base classes

The ASLM provides a number of base classes that force the v-table first (place it at the beginning of the object) and provide routines that give the user access to some of the objects' meta information (for example, in which library the object is implemented and the class IDs of the parents of the class). Some of the base classes also provide additional member functions that are commonly found in base classes such as `IsValid` and `Flatten`. All of these base classes override the `new` and `delete` operators so they use the ASLM global `new` and `delete` operators. For more information, see "Using the ASLM Global `new` and `delete` Operators" later in this chapter.

The ASLM does not force you to use any of these base classes for your exported classes. However, if you do not use them or use a base class that forces the v-table first, you will not be able to call `CastObject` or `CastObjectToMain` on instances of subclasses of your base class. You will also not be able to call any of the global routines that provide meta information. These routines start with "GetObjects" (for example, `GetObjectsClassID)` and are simply global versions of the member functions that are provided with ASLM base classes.

The original ASLM base class was the `TDynamic` class. It inherits from `SingleObject` and forces the v-table to be first by not having any data members and by providing at least one virtual function. `TDynamic` provides a number of pure virtual member functions such as `IsValid` and `Flatten` and also a number of inline member functions for accessing meta information about the class. `TDynamic` also provides the ability to have instances of its subclasses be registered with the Inspector. See "Registering C++ Objects with the Inspector" in Chapter 7, "ASLM Utilities," for more information on the Inspector. See "`TDynamic`" in Chapter 9, "Utility Classes and Member Functions," for details on the available `TDynamic` member functions.

The `TDynamic` class has many virtual functions which causes subclasses to have a larger v-table. The `TSimpleDynamic` base class was created to solve this problem by getting rid of all the virtual functions except for the destructor. This makes the v-table much smaller, but also means that you cannot use any of the `TDynamic` virtual functions and you cannot register `TSimpleDynamic` subclasses with the Inspector.

The `TDynamic` class also has the disadvantage of inheriting from `SingleObject`, so it can not be used with multiple inheritance. This problem was solved by adding the `TStdDynamic` base class, which is the same as `TDynamic` except that it does not inherit from `SingleObject`. Since `TStdDynamic` does not inherit from `SingleObject`, it does not have the simple v-table format and, therefore, its v-table is not as efficient. It also cannot have instances of its subclasses registered with the Inspector.

The `TStdSimpleDynamic` class combines the features of both `TStdDynamic` and `TSimpleDynamic`. It does not inherit from `SingleObject` and does not provide any additional virtual functions. Its v-table is small, but does not use the simple v-table format. Also, it cannot have instances of its subclasses registered with the Inspector.

Lastly `MDynamic` was created to be used as a mixin class for multiple inheritance. It does not provide any of the `TDynamic` member functions for accessing meta information. It only provides a virtual destructor to force the v-table first.

Using the ASLM global `new` and `delete` operators

The ASLM has its own global `new` and `delete` operators that allocate memory from pools. These are the same `new` and `delete` operators that are used for any class that subclasses `TDynamic`.

The header file GlobalNew.h declares the `new` and `delete` operators as follows:

```
void* operator new(size_t size, TMemoryPool*);
void* operator new(size_t);
void  operator delete(void*);
```

If a client includes the header file GlobalNew.h, the ASLM uses the global `new` and `delete` operators for all objects created and all memory allocated with the global `new` operator.

You can use `new` with a pool argument to allocate memory from a specific pool, or without a pool argument to allocate memory from the default memory pool (the default pool is explained in “Memory Management Classes” in Chapter 8, “ASLM Utility Class Categories.”)

WARNING The ASLM global `new` operator cannot be used by a client until `InitLibraryManager` has been called, but shared libraries may (and should) always use it.

You must make certain that an object is both created and deleted using the same `new` and `delete` operator pair since `delete` must know how `new` allocated the memory. You cannot mix the ASLM `new` and `delete` operators with the `new` and `delete` operators that are supplied with CPlusLib.o. If your application has to delete objects created by a shared library using the ASLM `new` operator, you must use the ASLM `delete` operator.

Shared libraries should never use the `new` and `delete` operators that are supplied with CPlusLib.o since they rely on the C library memory management when does not work well with shared libraries.

Allocating and freeing memory for an object is normally done in an object’s constructor and destructor which are implemented in the shared library. Thus the implementation of the library normally controls how objects are allocated and freed. However, when an overloaded `new` operator is used (such as the ASLM `new` operator that takes a memory pool parameter), the memory allocation is actually done in the client’s code. This means that if the client uses a `new` operator that is not compatible with the shared libraries `delete` operator, then the object’s destructor will not now how to properly free the memory and may crash. For this reason, and because the `new` and `delete` operators in CPlusLib.o do not work well with shared libraries, it is strongly advised that both clients and shared libraries always include GlobalNew.h so the ASLM global `new` and `delete` operators are always used.

LibraryManager.o and LibraryManager.n.o also contain implementations of the ASLM global `new` and `delete` operators. Basically these versions do the same thing as the inline versions in GlobalNew.h. They are useful when for some reason the code that calls `new` or `delete` cannot be compiled with GlobalNew.h. For example, when creating an array of objects with `new`, CFront generates code to allocate the memory for the array using the `new` operator and to call the constructor of each object in the array. This means that it uses the implementation of whichever global `new` operator it links with, even if you include GlobalNew.h. This causes problems if you are declaring the class in a shared library and you link with the global `new` operator in CPlusLib.o, which uses `calloc` to allocate the memory.

Link with LibraryManager.o first if you want to use the ASLM global `new` operator. Link with CPlusLib.o first if you want to use the default C++ global `new` operator. Be careful when linking LibraryManager.o first. If you try to create an object using `new` before calling `InitLibraryManager`, you will crash. Wait until you have called `InitLibraryManager` to perform operations such as using streams (`cout`, `cin`, and so on) that use `new` when they are first called. It is generally best to link LibraryManager.o first for shared libraries. Link either one first for applications, depending on whether the application needs to use the `new` operator before calling `InitLibraryManager`.

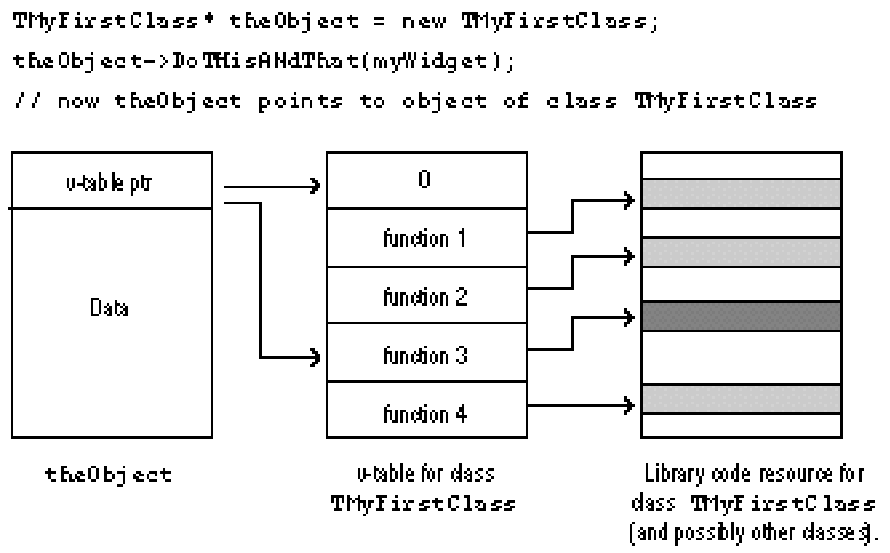
Virtual functions

In a C++ class, you can declare any member function to be a virtual function. In C++, a virtual function is called by a single indirection through a table of pointers to the functions. This table is called the *v-table*, or *virtual function table.*

In the ASLM implementation of virtual functions, a shared library contains the virtual functions implemented by one or more C++ classes. The v-table that is used to call the functions is built at run time so that references can be resolved when a shared library is dynamically loaded and linked.

There is only one copy of a shared class' v-table. It is stored in the global world of the shared library that implements that shared class. This help reduce memory footprint when multiple clients make use of the same shared class.

Figure 6-1 shows how the ASLM uses v-tables to call virtual functions.



```
TMyFirstClass* theObject = new TMyFirstClass;
theObject->DoThisAndThat(myWidget);
// now theObject points to object of class TMyFirstClass
```

v-table ptr
Data
0
function 1
function 2
function 3
function 4
theObject
v-table for class TMyFirstClass
Library code resource for class TMyFirstClass (and possibly other classes).

Figure 6-1 Virtual function tables

Since the call to a virtual function is indirect—through a pointer to an object—the code for the implementation of a virtual function does not have to be in the same code segment as the caller of the virtual function. In Figure 6-3, the implementation of `TMyFirstClass` is in a shared library. The method `theObject->DoThisAndThat` is a virtual function of the object called `theObject`. If `DoThisAndThat` is the third virtual function in the v-table shown in the diagram, then the highlighted code that implements function 3 is called.

V-table based function calls provide the fastest possible way to call a dynamically linked function. This is one of the benefits you get when exporting functions by implementing them as member functions of a shared class rather than as functions in a function set. It also shows an advantage of virtual member functions over non-virtual member functions for exported classes. Since non-virtual member function calls must go through a function stub just like function set function calls do, they are not as fast as virtual function calls.

In some instances virtual function calls do not go through the v-table. This includes virtual function calls for stack objects and calls you explicitly make to inherited functions. In these cases, the client object file must contain a stub for the virtual function or the client will not link. For more information, see “Creating Stack Objects” earlier in this chapter.

III Reference

7 ASLM Utilities

This chapter describes the ASLM utility functions that you can use to perform a number of tasks including:

- registering shared library files and folders
- preloading dependent libraries
- loading and unloading shared libraries
- client death watch notification
- setting up global worlds
- using the local library manager
- calling functions by name
- getting information about function sets
- using interrupts
- handling exceptions
- verifying an object’s type
- verifying a class’s base class
- loading and unloading the ASLM
- entering and leaving system mode

Registering shared library files and folders

Several utility functions allow you to register shared library files and folders in the following manner:

- You can register a folder as a shared library file folder and then place library files in the folder.
- You can register a file as a shared library file. In this case the shared library file can be located in any folder.

Registering and unregistering shared library file folders

You can make a shared library file accessible to clients by placing it in any folder that is registered as a shared library file folder. When you register a folder as a shared library file folder, the ASLM keeps track of shared library files that are dragged into and out of the folder. All shared library files that are in the folder are available to clients. You can drag library files into or remove library files from a registered folder at any time you choose. You can also rename or delete library files that are stored in the registered folder.

If you decide that you no longer want to use a folder as a registered folder, you can unregister it. The ASLM keeps a use count for all registered folders, so multiple users can register the same folder without fear of it becoming unregistered by another user.

Registering a shared library file folder

You can register a folder as a shared library file folder by calling the `RegisterLibraryFileFolder` function. The syntax of the `RegisterLibraryFileFolder` function is:

```
OSErr RegisterLibraryFileFolder(const TFileSpec*);
```

The `RegisterLibraryFileFolder` call takes a `TFileSpec` parameter that specifies the location of the folder being registered. (See "Specifying a Library File" later in this chapter for more information on `TFileSpec`.) Currently, `TMacFileSpec` is the only `TFileSpec` type that is supported. The ASLM returns a `kNotSupported` error if you pass the `RegisterLibraryFileFolder` function another type.

The `RegisterLibraryFileFolder` call returns a `kNoError` result if it is successfully executed, and returns `kFileNotFound` if it cannot find the specified directory. If the folder is already registered, a registered count for the folder is incremented. This prevents the folder from being unregistered if another user attempts to unregister it by calling `UnregisterLibraryFileFolder`.

Unregistering a shared library file folder

You can delete the registration of a folder—unregister the folder—by calling the `UnregisterLibraryFileFolder` function. The syntax of the `UnregisterLibraryFileFolder` function is:

```
OSErr   UnregisterLibraryFileFolder(const TFileSpec*,
                                    Boolean forceUnload);
```

`UnregisterLibraryFileFolder` takes a `TFileSpec` parameter that specifies the folder to unregister. When a folder is unregistered, the registered count for the folder is decremented. If the count has reached 0, the folder is actually unregistered. Otherwise, the folder remains registered. This procedure prevents the folder from being unregistered while it is still registered by another client.

If the count has reached 0 and one or more clients are still using a library file in the folder, the `kFolderInUse` error is returned. To avoid this error, all clients must do the following:

- Call `ResetFunctionSet(NULL)` if they have used any function sets. This forces a client to release a function set so that the shared library containing the function set can be unloaded. (This is done automatically when a client calls `CleanupLibraryManager`.)
- Explicitly close any library file they have opened before calling `UnregisterLibraryFileFolder`. A client opens library files when it calls `PreFlight` or `OpenLibraryFile` and closes library files when it calls `CloseLibraryFile`. (This is done automatically when a client calls `CleanupLibraryManager`.)

The `UnregisterLibraryFileFolder` function also accepts a `forceUnload` parameter. If the value of `forceUnload` is `true`, the `UnregisterLibraryFileFolder` function forces all loaded libraries in the folder to be unloaded, even if they are in use. It also forces all open instances of the library file to be closed. Therefore, the `kFolderInUse` error will never be returned. Unless you are certain that all loaded libraries in a registered folder can be safely unloaded, the value of the `forceUnload` parameter should be `false`. If `UnregisterLibraryFileFolder` is called with a `forceUnload` value of `false,` no library files that the specified folder contains are deleted until all the libraries in the folder are unloaded. If the folder's registered count has not reached 0, the `forceUnload` parameter has no effect.

How registered folders are tracked

If a registered folder is moved or renamed, the ASLM tracks the folder's new name and location. However, when you want to unregister the folder, you must specify its new name and location. For this reason it is best to use a `TMacFileSpec` that does not specify a folder name. The `TMacFileSpec` can specify `vRefNum` and `dirID`, since these remain the same even when the folder is moved or renamed.

Registering folders with the Inspector

The Inspector application that is shipped with the ASLM provides examples of how folders can be registered and unregistered. When Inspector is running, you can register and unregister folders by choosing commands from the Commands menu. The Inspector is described in Appendix B "ASLM Utility Programs."

Registering and unregistering shared library files

If you do not want to register a folder that contains a shared library file, you can register the shared library file that is inside the folder.

When you have registered an individual shared library file without registering the folder in which it resides, the ASLM can find the registered file and make it accessible to clients in the same way it would if it were placed in the System 7 Extensions folder, the System 6 System Folder, or a registered folder.

When an individual file is registered as a shared library file, it is available for any client to use; it is not private to the user that registered it.

The ASLM maintains a registered count on each registered shared library file so that more than one user can register a file without it becoming unregistered when just one user attempts to unregister it.

You can register a shared library file by calling the `RegisterLibraryFile` function. You can unregister a library file by calling `UnregisterLibraryFile` or `UnregisterLibraryFileByFileSpec`. The syntax for the `RegisterLibraryFile`, `UnregisterLibraryFile`, and `UnregisterLibraryFileByFileSpec` functions is:

```
OSErr RegisterLibraryFile(const TFileSpec*, TLibraryFile**);

OSErr UnregisterLibraryFile(TLibraryFile*, Boolean forceUnload);

OSErr UnregisterLibraryFileByFileSpec(const TFileSpec*,
                                      Boolean forceUnload);
```

The `RegisterLibraryFile` function takes a `TFileSpec` parameter that specifies the location of the library file being registered. Currently, `TMacFileSpec` is the only `TFileSpec` type that is supported. If the `RegisterLibraryFile` call is successful, the call returns a result of `kNoError` and a pointer to the `TLibraryFile` object that it has created. This `TLibraryFile` object is stored in the `TLibraryFile**` parameter. If you pass `NULL` in this parameter, the `TLibraryFile` object is not returned. If the ASLM cannot find or open the file, the `RegisterLibraryFile` function returns a result of `kFileNotFound`. If there is not enough memory to process the file, the call returns a result of `kOutOfMemory`. If the file is already registered, the ASLM increments the registered count for the file.

The `UnregisterLibraryFile` function takes a `TLibraryFile` parameter that specifies the file to unregister. This parameter should be the same as the `TLibraryFile` that was returned by `RegisterLibraryFile` when the file was registered.

The `UnregisterLibraryFileByFileSpec` function takes a `TFileSpec` parameter that specifies the file to unregister. Currently, `TMacFileSpec` is the only `TFileSpec` type that is supported.

When you call `UnregisterLibraryFileByFileSpec` or `UnregisterLibraryFile`, the ASLM decrements the registered count for the file. If a file's registered count has reached 0 when the function is called, the ASLM unregisters the file and deletes the file's associated `TLibraryFile` object. If the file's registered count is more than 0 when the function is called, the file remains registered.

Both `UnregisterLibraryFile` and `UnregisterLibraryFileByFileSpec` accept a `forceUnload` parameter. If the value of `forceUnload` is `true`, the functions force all loaded libraries in the file to be unloaded, even if they are in use. Therefore, unless you are certain that all loaded libraries in a registered file can be safely unloaded, the value of the `forceUnload` parameter should be `false`. If `forceUnload` has a value of `false`, the library file is not unregistered until all the libraries in the file are unloaded. If the file's registered count has not reached zero, the `forceUnload` parameter has no effect.

If you keep track of the `TLibraryFile` object returned by `RegisterLibraryFile`, you can unregister a file by calling `UnregisterLibraryFile`. You can also unregister a file by calling `UnregisterLibraryFileByFileSpec` and specify the `TFileSpec` of the file to be unregistered. This is useful if you want to let the user choose which file to unregister.

If you want to unregister a file by calling `UnregisterLibraryFile`, you should make sure that the file cannot be deleted, because that would cause the `TLibraryFile` object to be deleted, resulting in a crash later on when you call `UnregisterLibraryFile`. To prevent the library file from being deleted, simply call `OpenLibraryFile` after you register the file, and call `CloseLibraryFile` *after* you unregister the file. If your client is going to terminate after registering the library file, the client should call `OpenLibraryFile` and `CloseLibraryFile` while in system mode.

If a registered file is dragged into a registered folder or the folder that the file is in becomes registered, the file still maintains its identity as a registered file and is not unregistered even if its folder is unregistered.

You can register a file that is in a registered folder. It then remains registered even if its folder is unregistered. If you unregister a file that is in a registered folder by calling `UnregisterLibraryFile`, it remains registered (since it is still in a registered folder).

The Inspector application that comes with the ASLM provides examples of how files can be registered and unregistered. When the Inspector is running, you can register and unregister shared library files by choosing commands from the Commands menu.

Preloading all dependent libraries

The MPW tool `CreateLibraryLoadRsrc` that is provided with the ASLM, creates a resource for preloading all libraries that a client depends on. To use the `CreateLibraryLoadRsrc` tool, you must link your client or library using the `-map` option, which causes a link map to be generated. The `CreateLibraryLoadRsrc` tool creates a resource of type `'libi'` in source code form that you can `Rez` into your application or shared library. This `'libi'` resource is used by the ASLM routines `LoadLibraries` and `UnloadLibraries`, described in “Loading and Unloading Shared Libraries” later in this chapter. The `'libi'` resource contains information about which function sets and classes the client is dependent on. It does not include dynamic dependencies (such as, those created using `NewObject` or `GetFunctionPointer`).

The `BuildSharedLibrary` and `LinkSharedLibrary` scripts automatically invoke the `CreateLibraryLoadRsrc` tool to create a `'libi'` resource for each library that they create, so generally only non-library writers need to explicity use this tool.

The syntax of the `CreateLibraryLoadRsrc` command is:

```
CreateLibraryLoadRsrc -map <MapFileName> -o <Output .r file name>
      [-p] [-v] [-a] [-resid #] [-not <class>] [-only <class>]
```

where:

`-p`

This option writes a progress report.

`-v`

This option writes verbose output.

`-a`

This option causes the resource information to be appended to the output .r file instead of overwriting the output file.

`-resid #`

This option forces the resource ID number of the `'libi'` resource. You should not normally use this switch. It is used by the `BuildSharedLibrary` and `LinkSharedLibrary` scripts when they create shared libraries. Clients such as applications or tools that call `InitLibraryManager` must leave the resource ID number at 0.

`-not <class>`

This option lets you specify function sets and classes that you do not want included in the `'libi'` resource. It can be used multiple times on the command line.

`-only <class>`

This option lets you specify that only a particular function set or class should be included in the `'libi'` resource. It can be used multiple times on the command line.

Loading and unloading shared libraries

Shared libraries load and unload automatically as you use them. However, you may want to explicitly load a shared library so it can be used at interrupt time or so that you do not have to worry about exception handling if the shared library cannot be loaded when needed. The following routines help provide further control over loading and unloading shared libraries:

```
OSErr      LoadClass(const TClassID, BooleanParm forceAll);
OSErr      UnloadClass(const TClassID);
Boolean    IsClassLoaded(const TClassID);

OSErr      LoadFunctionSet(const TFunctionSetID, BooleanParm forceAll);
OSErr      UnloadFunctionSet(const TFunctionSetID);
Boolean    IsFunctionSetLoaded(const TFunctionSetID);

OSErr      LoadLibraries(BooleanParm forceAll, BooleanParm doSelf);
OSErr      UnloadLibraries(void);
void       ResetFunctionSet(const TFunctionSetID);
```

`IsFunctionSetLoaded`
`IsClassLoaded`

Use the `IsFunctionSetLoaded` and `IsClassLoaded` functions to check whether the function set or class is loaded. The `IsFunctionSetLoaded` function returns `true` if the implementation of the specified function set ID is loaded. The `IsClassLoaded` function returns `true` if the implementation of the specified class ID and all of its parent classes are loaded.

The `IsFunctionSetLoaded` and `IsClassLoaded` functions indicate if the library implementing the function set or class (and the class's parents) is loaded, but give no indication of whether or not the code segments of the library or any other libraries that the library depends on are loaded. There are two ways to ensure that all code segments and all dependent libraries are also loaded. The first is to call `LoadFunctionSet` or `LoadClass` and pass in `true` for the `forceAll` parameter. The second way is to make sure that all the dependent libraries are built with `flags=segUnload` (the default) and the library in which the class or function set is implemented is built with `flags=forcedeps` and `flags=segUnload.`

LoadLibraries
UnLoadLibraries

The CreateLibraryLoadRsrc function, described in “Preloading All Dependent Libraries,” earlier in this chapter can create a 'libi' resource that describes all of the function sets and classes that a client or shared library uses.

The LoadLibraries function reads the caller’s 'libi' resource and then calls LoadFunctionSet to load the function sets and LoadClass to load the classes described in the 'libi' resource.

For non-library clients, LoadLibraries reads the 'libi' #0 resource. In this case, the 'libi' resource must be created and Rezed into your client using the CreateLibraryLoadRsrc tool described under the previous heading.

For shared libraries, LoadLibraries reads the 'libi' resource that has the same resource ID as the 'libr' resource for the library. In this case, the resource is created and Rezed into your shared library automatically by the BuildSharedLibrary and LinkSharedLibrary scripts.

WARNING LoadLibraries is not interrupt-safe.

When you call LoadLibraries, the forceAll parameter is used to force all of the code segments belonging to the dependent libraries to load. It is the same as the forceAll parameter passed to LoadFunctionSet and LoadClass .

The doSelf parameter is used only for libraries. If doSelf is true, it forces the library to load itself. This prevents the library from unloading until the library makes an explicit UnloadLibraries call, even if the library has no clients. If a false doSelf parameter is passed, the library unloads when it has no clients, and an UnloadLibraries call is made automatically.

You can pass a true doSelf parameter to LoadLibraries when a library is preloaded (has its preload flag set) and you want to make sure that the library stays loaded, even if it has no clients. In this situation, you normally call LoadLibraries from your library’s Initproc. Remember that a library that is preloaded will immediately unload unless it keeps itself loaded. For example, a library can keep itself loaded by calling LoadLibraries. You can get similar results by setting the library’s stayLoaded flag (described in “Library Declaration” in Chapter 5, “Writing and Building Shared Libraries.”)

You can pass a `false` `doSelf` parameter when a library must make sure that all of the other libraries that it depends on are loaded, but still requires them to be unloaded when it has no clients. Once again, you normally call `LoadLibraries` from your library's `Initproc`, but a better alternative is to set the library's `loaddeps` flag or `forcedeps` flag (described in "Library Declaration" in Chapter 5, "Writing and Building Shared Libraries.")

The `LoadLibraries` function returns an error code if it cannot find any of the dependent libraries that it requires (or if it cannot load them if it is requested to do so). It also returns an error if it cannot find or load the `'libi'` resource that it requires.

You can instruct the shared library to call `LoadLibraries` when your library is loaded by setting your library's `loaddeps` flag, `forcedeps` flag, or `stayLoaded` flag. All these flags cause `LoadLibraries` to be called, but `forcedeps` also causes a `forceAll` parameter of `true` to be passed, and `stayLoaded` causes a `doSelf` parameter of `true` to be passed. If you set the `stayLoaded` flag to `true`, your library must explicitly call `UnloadLibraries` to be unloaded.

The `UnloadLibraries` function calls `UnloadFunctionSet` or `UnloadClass` for every function set or class loaded by `LoadLibraries`. It also clears out any cached information in the caller for any library that was being used and was unloaded by the call to `UnloadLibraries`.

It is not necessary to call `UnloadLibraries` unless `LoadLibraries` was called with `true` passed to the `doself` parameter. When a client calls `CleanupLibraryManager` or a library is being unloaded, `UnloadLibraries` is automatically called to unload any libraries that have been loaded by a `LoadLibraries` call.

LoadClass
UnloadClass

The `LoadClass` function loads the shared library or shared libraries needed for the implementation of a specified class ID. The ID of the class to be loaded is passed to `LoadClass` as a parameter. If a class depends on other classes in other shared libraries, those shared libraries are also loaded. If the required libraries are already loaded, `LoadClass` increments their use counts. The `LoadClass` method returns `kNoError` if the specified class and all dependent classes are successfully loaded. If the call is unsuccessful, an error is returned. If the `forceLoad` parameter is set to `true`, all the code segments of the target libraries are loaded. This procedure ensures that interrupt-safe calls can be made to the specified shared library.

When `LoadClass` is called, all dependencies of the library are loaded, not just the parent classes. The only exceptions are dependencies created by functions that are called by name, or objects that are created by calling `NewObject`.

The ASLM keeps track of all `LoadClass` calls and calls `UnloadClass` automatically when a client calls `CleanupLibraryManager`. Therefore, it is not necessary to balance `LoadClass` calls with calls to `UnloadClass`. However, you should still call `UnloadClass` when you have finished using a class. By doing so, you can make sure that the class library is unloaded if the library is no longer in use and you do not plan to call `CleanupLibraryManager` soon (for example, when the `LoadClass` call is the only thing keeping the library loaded).

WARNING `LoadClass` is not interrupt-safe.

The `UnloadClass` function, unlike `LoadClass`, *is* interrupt-safe. The `UnloadClass` function returns `kNoError` if the specified class and all dependent classes are successfully unloaded. If the call is unsuccessful, an error is returned.

The `UnloadClass` function returns `kNotAllowedNow` if the current client has not made a corresponding `LoadClass` call, and returns `kNotFound` if the specified `TClassID` object is not a valid class ID.

LoadFunctionSet UnloadFunctionSet

`LoadFunctionSet` and `UnloadFunctionSet` work exactly like `LoadClass` and `UnloadClass`, except they are used to load and unload a function set instead of a class. `LoadFunctionSet` loads the shared library or shared libraries needed for the implementation of a specified function set. The ID of the function set to be loaded is passed to `LoadFunctionSet` and `UnloadFunctionSet` as a parameter.

WARNING `LoadFunctionSet` is not interrupt-safe.

ResetFunctionSet

`ResetFunctionSet` clears all cached information in the client's function stubs for the specified function set. When a function in a function set is called for the first time, the function stub linked with the caller looks up the address of the function and places the address in its cache. This process causes the function set's library to be loaded if it is not already loaded, and also increments the library's use count. The only way to decrement the library's use count and cause the library to be unloaded is to call `ResetFunctionSet`, passing it the `TFunctionSetID` of the function set that you want to reset. This causes all cached information for the function set in the client's function stubs to be cleared out, allowing the library's use count to be decremented. If the library's use count is decremented to zero, the library is unloaded.

You can reset all function sets that a client uses by passing `NULL` to `ResetFunctionSet`. All function sets are reset automatically when the client quits (by calling `CleanupLibraryManager`) or unloads (in the case of a library).

Client death watch notification

The ASLM provides a notification facility that you can use to determine when a client *goes away*. A client goes away when it calls `CleanupLibraryManager` or when a shared library unloads (since shared libraries are also considered clients).

To keep track of when clients go away, you can register a death watch notifier, also called a *death watcher*. To register a death watcher, you can call the `InstallDeathWatcher` function. When you no longer want to be notified of clients that have gone away, you can call the `RemoveDeathWatcher` function.

There are several reasons for installing a death watcher. For example, you may have written an application or library that makes callbacks to its clients when certain conditions exist. By maintaining a death watcher, you can avoid attempting to make a callback to a client that has gone away.

Another reason for installing a death watcher is to make sure that your application is notified when it is going away (probably because it has crashed). The Inspector application provides an example of using death watchers in this manner. Before going away, Inspector makes sure certain objects are deleted. The Inspector application is described in Appendix B "ASLM Utility Programs."

How death watchers work

The syntax of `InstallDeathWatcher` and `RemoveDeathWatcher` is:

```
Boolean InstallDeathWatcher(TNotifier* notifier);
Boolean RemoveDeathWatcher(TNotifier* notifier);
```

Both `InstallDeathWatcher` and `RemoveDeathWatcher` take a `TNotifier` object as a parameter. When `InstallDeathWatcher` has been called, the specified `TNotifier` object's `Notify` function is called each time a client goes away.

For more information on the `TNotifier` class and its member functions, see Chapter 9, "Utility Classes and Member Functions."

The `InstallDeathWatcher` call returns `true` if it is successfully executed; otherwise, it returns `false` . However, `InstallDeathWatcher` cannot fail unless the ASLM runs out of memory—which is unlikely.

The `Notify` function

When the specified `TNotifier` object's `Notify` function is called, the method's `notifyData` parameter contains a pointer to the `TLibraryManager` object of the client that is going away, and the method's `EventCode` parameter contains `kDeathEvent`.

If a client is being notified about its own death, the `TLibraryManager` pointer that is passed to its `TNotifier` object's `Notify` function is the same as the one returned by `GetLocalLibraryManager`.

When the specified `TNotifier` object's `Notify` function is called, the method's `OSErr` parameter contains one of three values: `kNoError` if a client called `CleanupLibraryManager`, `kCodeNotLoaded` if a library is being unloaded, and `kLibraryManagerNotLoaded` if the ASLM is being unloaded. You never have to worry about a `kLibraryManagerNotLoaded` error code unless you want to add debugging code to your client so it can handle the ASLM being reloaded from the Inspector application or from an explicit `UnloadLibraryManger` call in your own code (which should be there for debugging purposes only). The Inspector does check for the `kLibraryManagerNotLoaded` error code, providing an example of this kind of checking.

Global world functions

The ASLM provides a number of routines for setting up a client's global world:

```
GlobalWorld    GetGlobalWorld();
GlobalWorld    OpenGlobalWorld();
void           CloseGlobalWorld(GlobalWorld oldWorld);
GlobalWorld    SetCurrentGlobalWorld(GlobalWorld newWorld);
GlobalWorld    GetCurrentGlobalWorld(void);
```

The `GetCurrentGlobalWorld` and `SetCurrentGlobalWorld` functions deal with the current global world setting. They are the same as the Macintosh `GetA5` and `SetA5` routines and are used to get and set the current global world, which is represented by the A5 register on the Macintosh.

The `OpenGlobalWorld`, `CloseGlobalWorld`, and `GetGlobalWorld` functions deal with the global world belonging to a library or model far client. Thus, the global world returned by `GetGlobalWorld` may not be the same as the current global world.

The `GetGlobalWorld` function returns the global world pointer for the client making the call. The global world returned by `GetGlobalWorld` may not be the same as the current global world. Use `GetGlobalWorld` to get the global world for the library or application client making the call. This can be useful if you need to pass the global world to code that may need to set it at a later time.

The `OpenGlobalWorld` function simply calls `GetGlobalWorld` and passes the result to `SetCurrentGlobalWorld`. The `CloseGlobalWorld` function performs the same operation as `SetCurrentGlobalWorld`, except that it does not return a result.

The `CloseGlobalWorld` function reverts to the global world that was current before calling `OpenGlobalWorld`. When you call `CloseGlobalWorld`, you must pass it the global world that was returned by `OpenGlobalWorld`. It is the same as calling `SetCurrentGlobalWorld(oldWorld)` except that it does not return a global world.

You can call `EnterSystemMode` to make the ASLM global world the current global world. Although there is generally no reason to make the ASLM global world the current global world, you should be aware that this is a side effect of calling `EnterSystemMode`. If you want to enter system mode but do not want the current global world changed, call `GetCurrentGlobalWorld` before calling `EnterSystemMode` and pass the result to `SetCurrentGlobalWorld` after calling `EnterSystemMode`. For additional information on `EnterSystemMode`, see “Entering and Leaving System Mode” later in this chapter.

Since libraries are always compiled with model far, it is not necessary to call `OpenGlobalWorld` before using globals or making intersegment calls.

Note: Only libraries and model far clients should call `GetGlobalWorld`, `OpenGlobalWorld`, and `CloseGlobalWorld`.

Support for stand-alone code resources

A number of routines are provided to make it easier to set up a global world for stand-alone code resources and make the code resource the current client. (These routines are called by stand-alone code only.)

```
OSErr          InitGlobalWorld(void);
void           FreeGlobalWorld(void);
OSErr          InitCodeResource(void);
void           EnterCodeResource(void);
void           LeaveCodeResource(void);
```

The `InitGlobalWorld` function creates and initializes the global world for stand-alone code on the Macintosh Operating System—for example, INITs and CDEVs. It also calls `SetCurrentGlobalWorld`. The `FreeGlobalWorld` function frees the memory used by the global world created by `InitGlobalWorld`.

The `InitCodeResource` function calls `InitGlobalWorld` to set up a global world for code resources and to store the pointer to the global world in a PC-relative location so that it can be used later.

The `EnterCodeResource` function is used to set the global world of code resources as the current global world and to make the code resource the current client. It uses the global-world pointer saved by `InitCodeResource`. It is most useful when the code resource only calls `InitLibraryManager` once but may be reentered multiple times before calling `CleanupLibraryManager`. The `LeaveCodeResource` function will undo what `EnterCodeResource` does. These two routines are *not* reentrant.

When initializing the code resource, you should do the following:

```
GlobalWorld savedWorld = GetCurrentGlobalWorld();
InitCodeResource();
InitLibraryManager();
/* Do anything else you want before returning. If */
/* you make ASLM calls then you must also use */
/* Enter/LeaveCodeResource.
SetCurrentGlobalWorld(savedWorld);
```

Each time you reenter the code resource, you should do the following:

```
EnterCodeResource()
/* do ASLM related stuff */
LeaveCodeResource()
```

When you have finished, do the following:

```
EnterCodeResource()
CleanupLibraryManager();
LeaveCodeResource();
```

Creating and deleting the local library manager

All clients of the ASLM are required to have a `TLibraryManager` object installed. This `TLibraryManager` object is referred to as the *local library manager*. For shared libraries, the local library manager is created and installed when the library is loaded and is deleted when the library is unloaded. For non-library clients, the local library manager is created by `InitLibraryManager` and is deleted by `CleanupLibraryManager` (discussed in Chapter 4).

The local library manager is used behind the scenes when clients make many ASLM calls including calling functions in a function set and creating C++ objects when there is no information cached about the function or C++ class. It is also used when calling most of the ASLM utility functions. It provides the link between the client and the ASLM.

The local library manager is also used by the ASLM to *represent* a client. As explained in “The Current Client,” in Chapter 4, the local library manager is returned from and passed as a parameter to the functions that are used to set and get the current client.

The local library manager is also used as the *keeper* of the client’s local pool pointer. See “Memory Management Classes” in Chapter 8, “ASLM Utility Class Categories” for more information on the local pool.

The `InitLibraryManager` function

When a non-library client wants to create an object that is implemented in a shared library, or wants to use a function that is implemented in a shared library, the client must call `InitLibraryManager` first. The `InitLibraryManager` function creates a local instance of the `TLibraryManager` class (which can be accessed by calling `GetLocalLibraryManager`).

The `InitLibraryManager` function is declared in the LibraryManager.h file as follows:

```
#ifdef __cplusplus
      OSErr    InitLibraryManager(size_t poolsize = 0,
                                  ZoneType = kCurrentZone,
                                  MemoryType = kNormalMemory);
#else
      OSErr    InitLibraryManager(size_t poolsize, short zoneType,
                                 short memType);
#endif
```

In the above declaration, `InitLibraryManager` creates a local memory pool of size `poolsize.` The memory for the pool is obtained from the zone of type `ZoneType` and is of type `MemoryType.` The `ZoneType` and `MemoryType` parameters are declared in the LibraryManager.h file and are explained in Chapter 9, “Utility Classes and Member Functions.”

Non-C++ users do not need to be concerned with `InitLibraryManager` parameters unless they are making calls to C++ code that may want to allocate objects or memory out of the client’s local pool. Normally non-C++ users should just pass 0 for `poolsize`, `kCurrentZone` for `ZoneType`, and `kNormalMemory` for `MemoryType`. However, clients that make use of the ASLM at interrupt time should pass `kHoldMemory` for `MemoryType`.

The pool that `InitLibraryManager` creates serves as the local library manager’s object pool, which is the pool used to allocate memory for objects that are created using `NewObject`. You can access the pool by calling `TLibraryManager::GetObjectPool`. The pool is also called the local memory pool and can be accessed by calling `GetLocalPool`.

The `InitLibraryManager` function always creates a local memory pool, even if you pass it a pool of size zero (0). An object of class `TPoolNotifier` is attached to the pool so that the pool can grow instead of returning an error if it runs out of memory. The `TPoolNotifier` class can assist in automatically “growing” a pool when the pool comes dangerously close to running out of memory.

When `InitLibraryManager` creates a `TLibraryManager` object, the new `TLibraryManager` object and the new `TPoolNotifier` object are allocated from the local pool. The overhead for these two classes is added to the pool size passed to `InitLibraryManager`.

For more information on memory pools and the `TPoolNotifier` class, see Chapter 9, "Utility Classes and Member Functions."

WARNING The `InitLibraryManager` call is not interrupt-safe. You must be in your global world to call it.

The `CleanupLibraryManager` function

When you finish using the ASLM, you must call `CleanupLibraryManager`. The `CleanupLibraryManager` function deletes the local `TLibraryManager` object, its initial local pool (the pool created by `InitLibraryManager`), and the pool's `TPoolNotifier`. It also does some other house cleaning, including releasing any function sets that were used and closing any library files that were explicitly opened. Any `LoadClass`, `LoadFunctionSet`, and `LoadLibraries` calls are also undone. `CleanupLibraryManager` is only called by clients that called `InitLibraryManager`. Shared libraries should never call `CleanupLibraryManager`.

The `CleanupLibraryManager` function is declared in the header file LibraryManager.h. as follows:

```
void     CleanupLibraryManager();
```

The `CleanupLibraryManager` function is called automatically for application clients that either crash or do not call it before quitting. This means that if an application crashes, it releases any function sets it was using and closes any library files that it explicitly opened. However, it does not release any classes for which all instances were not deleted. The libraries that these classes are in remain loaded and in use until the computer is restarted.

WARNING The `CleanupLibraryManager` function is not interrupt-safe, and you must be in your global world when you call it.

Getting the local library manager

The function `GetLocalLibraryManager` returns the currently installed local library manager. For shared libraries, the local library manager is created and installed when the library is loaded and is deleted when the library is unloaded. For non-library clients, the local library manager is created by `InitLibraryManager`. The `GetLocalLibraryManager` function returns `NULL` if `InitLibraryManager` failed or has not been called yet. It is declared as follows:

```
TLibraryManager*    GetLocalLibraryManager;
```

A client can call `GetLocalLibraryManager` to test whether `InitLibraryManager` has been called successfully.

For more information on `InitLibraryManager` and the local library manager, see "Creating and Deleting the Local Library Manager" above.

Calling functions by name

The ASLM supports exporting and calling C functions by name. To make use of this capability, you must modify your client's .exp file. Any functions in your function set that you want to be exported by name should be preceded by the keyword `extern`. You can then call the `GetFunctionPointer` to obtain a pointer to the function. Of course, you can call the function in the usual manner as well.

The ASLM also supports calling functions by specifying the function's index in the function set. The `GetIndexedFunctionPointer` function is used for this and does not require that the function name be preceded by the `extern` keyword in the .exp file.

The syntax of these two functions is:

```
ProcPtr     GetFunctionPointer(const TFunctionSetID,
                               const char* funcName, OSErr*);

ProcPtr     GetIndexedFunctionPointer(const TFunctionSetID,
                                      unsigned int index, OSErr*);
```

The `GetFunctionPointer` function returns a pointer to a function, and takes the name of the function in the function set. The `GetIndexedFunctionPointer` function returns a pointer to a function, and takes the index of the function in the function set. The `TFunctionSetID` parameter is the ID of the function's function set and `funcName` is the name of the function.

If an error occurs while `GetFunctionPointer` or `GetIndexedFunctionPointer` is trying to obtain a function pointer, the call returns `NULL` and the appropriate error code is placed in the `OSErr*` parameter.

One possible use for `GetFunctionPointer` is to extend scripting languages. You can let the user specify the name of the function (and perhaps even the function set ID), and you can then call `GetFunctionPointer` to obtain the implementation of the function. This is similar to the way XCMDs work in HyperCard.

Another possible use for `GetFunctionPointer` or `GetIndexedFunctionPointer` is to allow the same routine to be implemented in more than one function set, with the option of choosing which function set is used. If you call the function directly, the function set whose client object (.cl.o) file you have linked with determines which function set is used. Using `GetFunctionPointer` or `GetIndexedFunctionPointer` allows you to choose at run time which function set to use.

By placing an `interfaceID=FunctionSetID` line in a client's export (.exp) file, you can associate a function set with an interface. It allows you to specify a common interface for your function sets that implement the same functions. You can then create multiple function sets with the same interface ID, and you can use the `GetFunctionSetInfo` function to find all such function sets with the same interface ID and then use `GetFunctionPointer` or `GetIndexedFunctionPointer` to get the correct function in the correct function set. All that is required is that each function set implement the same functions and that any given function has the same interface and either the same index or the same name in each function set. The `GetFunctionSetInfo` function is described in the next section.

The following is an example of a declaration of a function set that exports a function by name:

```
FunctionSet ExampleFSet
     {
          id = kExampleFunctionSet;
          exports = extern Hello;
     };
```

This example shows how you might call a function named `Hello` by name.

```
ProcPtr helloPtr;
helloPtr = GetFunctionPointer(kExampleFunctionSet, "Hello",
                              NULL);
*helloPtr();
```

The above example does not include error checking, which should be added. It may also be necessary to cast the result of `GetFunctionPointer` to a different function pointer.

WARNING Although `GetFunctionPointer` and `GetIndexedFunctionPointer` cause the shared library implementing the function to be loaded, they do not increment the shared library's use count. This means that unless something else is done to increment the library's use count, it will be unloaded the next time `SystemTask` is called. If you call the function returned immediately after the call to `GetFunctionPointer` or `GetIndexedFunctionPointer` then you do not have to worry (unless the function allows `SystemTask` to be called). However, if you plan on using the function pointer returned at a later time, you normally will call `LoadFunctionSet` immediately before or after the `GetFunctionPointer` or the `GetIndexedFunctionPointer` call and then call `UnloadFunctionSet` when you are done with the function pointer. This will ensure that the shared library stays loaded until you are done with the function pointer.

Getting information about function sets

The `GetFunctionSetInfo` function, the C interface to the `TClassInfo` class, is used to provide information about a function set or a series of function sets that have a common interface ID. In the latter case it is used to iterate over all function sets with the given interface ID. The `GetFunctionSetInfo` function returns a `TFunctionSetInfo` structure that is passed to other routines to get information about the function set. You free up `TFunctionSetInfo` by calling `FreeFunctionSetInfo`. The following routines are used in conjunction with `GetFunctionSetInfo` (these routines are C versions of the `TClassInfo` member functions):

```
TFunctionSetInfo* GetFunctionSetInfo(TFunctionSetID, OSErr*);

void              FreeFunctionSetInfo(TFunctionSetInfo*);

void              FSInfoReset(TFunctionSetInfo*);

TFunctionSetID    FSInfoNext(TFunctionSetInfo*);
```

```
Boolean          FSInfoIterationComplete(TFunctionSetInfo*);
TFunctionSetID   FSInfoGetFunctionSetID(TFunctionSetInfo*);
TFunctionSetID   FSInfoGetParentID(TFunctionSetInfo*, size_t idx);
TLibrary*        FSInfoGetLibrary(TFunctionSetInfo*);
TLibraryFile*    FSInfoGetLibraryFile(TFunctionSetInfo*);
unsigned short   FSInfoGetVersion(TFunctionSetInfo*);
unsigned short   FSInfoGetMinVersion(TFunctionSetInfo*);
```

After calling `GetFunctionSetInfo`, you can pass the `TFunctionSetInfo` object to any of the other routines. Most of them provide information about the current function set. Others are used to iterate over all function sets with the specified interface ID, if you passed an interface ID to `GetFunctionSetInfo` instead of an actual function set ID.

The first function set looked at is always the one specified when you called `GetFunctionSetInfo`. If you specified an interface ID, you need to call `FSInfoNext` to start iterating over all the function sets that have the specified interface ID. You can continue calling `FSInfoNext` until it returns `NULL`, gathering information about each function set as you proceed. The `FSInfoNext` function changes the function set being looked at to the next function set and returns the `TFunctionSetID` object of the next function set.

If you allowed calls to `SystemTask` or `WaitNextEvent` while iterating over the function set, the `TFunctionSetInfo` object may become invalid if the user has added or removed a shared library file from a registered folder. In this case, `FSInfoNext` will return `NULL` and `FSIterationComplete` will return `false`. If this happens you can call `FSInfoReset` and start the iteration over.

Use `FSInfoGetFunctionSetID` to get the `TFunctionSetID` of the current function set.

Use `FSInfoGetParentID` to get the interface ID of the current function set. The `idx` parameter should always be 0 and is there for historical reasons. A better name for this function is `FSInfoGetInterfaceID`, however, it is named `FSInfoGetParentID` for historical reasons.

The `FSInfoGetLibrary` function returns the `TLibrary` object in charge of the library that the current function set is implemented in. The `FSInfoGetLibraryFile` function returns the `TLibraryFile` object in charge of the library file that the current function set's library is in. Both the `TLibrary` and `TLibraryFile` objects have uses in other ASLM routines.

The FSInfoGetVersion function returns the version of the current function set and the FSInfoGetMinVersion function returns the minimum version that the current function set supports.

There is an example of how to use GetFunctionSetInfo on the *ASLM Examples* disk in the FunctionSetInfo folder.

Note: You can use TClassInfo to iterate over function sets and the routines mentioned above to iterate over classes.

Interrupt support

For some parts of the ASLM to work properly at interrupt time—for example, for memory to be allocated from memory pools and for objects to be created—the ASLM must be aware that the procedures are being executed during interrupts. You can call the EnterInterrupt function to inform the ASLM that you are executing code at interrupt time, and you can call LeaveInterrupt when you have finished. The AtInterruptLevel function returns true if EnterInterrupt has been called without a matching LeaveInterrupt call. Otherwise it returns false.

If your code is being executed because you have scheduled an operation on an ASLM scheduler such as TTimeScheduler or TIterruptScheduler, the ASLM is already aware that you are executing at interrupt time, so there is no need to call EnterInterrupt. (For more information on the ASLM scheduler classes, see “Process Management Classes” in Chapter 8, “ASLM Utility Class Categories.”)

You do not have to call EnterInterrupt before you schedule an operation on a TInterruptScheduler. This means that if your interrupt code only puts a TOperation on a TInterruptScheduler, you never have to call EnterInterrupt. However, if you use the new operator to allocate memory for the TOperation or any other object, you do need to call EnterInterrupt. The ASLM also provides an InInterruptScheduler function that can tell you if the TInterruptScheduler is currently running.

Virtually all ASLM calls and member functions of classes provided by the ASLM are interrupt-safe, with these exceptions:

- InitLibraryManager and CleanupLibraryManager
- LoadLibraryManager and UnloadLibraryManager
- InitCodeResource, InitGlobalWorld, and FreeGlobalWorld
- routines to load and unload library code segments
- RegisterLibraryFileFolder and UnregisterLibraryFileFolder

- `RegisterLibraryFile`, `UnregisterLibraryFile`, and `UnregisterLibraryFileByFileSpec`
- calls to `TMemoryPool::AddMemoryToPool` or `TMemoryPool::DownsizePool` to create a new pool or add memory to a pool. (`AddMemoryToPool` and `DownsizePool` return errors if they are called at interrupt time.)
- `TLibraryFile` resource management calls
- toolbox and operating system calls that are not interrupt-safe because they move or purge memory
- calls that cause a library to be loaded by creating an object (including stack objects)
- calls to a function in a function set that is not already loaded
- calls to `GetFunctionPointer` or `GetIndexedFunctionPointer` for a function in a function set that is not already loaded
- `LoadClass`, `LoadFunctionSet`, or `LoadLibraries`

You can verify that a class is loaded by calling `IsClassLoaded`. You can load a library while your client is executing in the foreground by calling `LoadClass`, `LoadFunctionSet`, or `LoadLibraries`. This will allow you to safely use the library at interrupt time.

EnterInterrupt
LeaveInterrupt

These functions should be called when you are in an interrupt service routine or a deferred task and you want to do something that will cause ASLM code to be executed such as allocating pool memory or creating an object. The ASLM needs to know that it is at interrupt time so it does not do anything harmful like trying to allocate Macintosh Memory Manager memory or load library code. This does not mean that all ASLM calls are safe at interrupt time, just that the ones that claim to be safe will only be safe if you do an `EnterInterrupt` call first.

You do not need to use these routines when your interrupt service routine is scheduling an operation on a `TInterruptScheduler`, when the operation gets executed at deferred task time, or when a `TTimeScheduler` operation gets executed. In the former case the ASLM realizes that you are at interrupt time and in the later two cases the ASLM does an `EnterInterrupt` before calling your operation and a `LeaveInterrupt` when your operation returns.

```
void EnterInterrupt(void);
```

```
void LeaveInterrupt(void);
```

`AtInterruptLevel`

This function returns `true` if you are currently executing at non-System Task time.

```
Boolean AtInterruptLevel(void);
```

`InInterruptScheduler`

This function returns `true` if you are currently running an interrupt scheduler.

```
Boolean InInterruptScheduler(void);
```

Exception handling

The ASLM provides exception handling macros that are used to catch exceptions that may be raised. Exceptions are raised by calling the `RAISE`, `Fail`, `FailNull`, `DebugFail`, and `DebugFailNull` functions (described later in this section). The only time the ASLM raises an exception is if it fails to load a shared library or fails to load a shared library's code segment after the shared library has already been loaded.

The ASLM will never raise an exception when calls are made that could return an error code instead, such as `LoadFunctionSet`. The strategy used is that if something useful cannot be done, such as returning an error code, an exception must be raised. For example, an exception is raised if a shared library cannot be loaded when a class is created or a function in a function set is called. The most common reason a library would fail to load is either it cannot be located or there is not enough memory for it.

Another reason an exception might be raised is if a library is loaded, but not all of its code segments are loaded. If a call is made to a member function or function implemented in an unloaded code segment and the code segment cannot be loaded because there is not enough memory, an exception is raised.

Of course shared libraries that a client uses may also raise exceptions for other reasons, but this is up to the developer of the shared library.

How to avoid raising exceptions

All shared libraries and clients must guard against raising exceptions. One way is to make sure that a library is loaded, along with all of its code segments, before trying to use it. You can use the `LoadClass`, `LoadFunctionSet`, and `LoadLibraries` functions for this. You can also specify certain flags when declaring a library in the library's .exp file that make sure all libraries that the library depends on are loaded when the library is loaded. If the libraries it depends on cannot be loaded, the library will fail to load.

The other way of guarding against raising exceptions is to use exception handling macros to catch any exceptions that are raised.

Exception handling macros

The ASLM exception handling macros match the DCE standard and are usable from C. Here is the syntax for handling exceptions:

```
TRY
     try_block
[CATCH (errcode)
     handler_block] ...
[CATCH_ALL
     handler_block]
[FINALLY
     final_block]
ENDTRY
```

The following macros are used for exception handling. They all conform to the DCE standard for exception handling.

- `TRY` starts a block of code that may end up raising an exception that you want to catch.

  A `try_block` or a `handler_block` is a sequence of statements, the first of which may be declarations, as in a normal block. If an exception is raised in the `try_block`, the catch clauses are evaluated to see if any match the current exception.

- `CATCH(errcode)` catches `errcode` if it is raised, and `CATCH_ALL` catches anything that `CATCH` has not caught.

  The `CATCH` or `CATCH_ALL` clauses absorb an exception; they catch an exception propagating out of the `try_block`, and direct execution into the associated `handler_block`. By default, the exception stops propagating. Within the lexical scope of a handler, it is possible to explicitly cause the same exception to resume propagating (this is called reraising the exception). It is also possible to raise a new exception.

- `RERAISE` reraises an exception that has been caught.

  The `RERAISE` statement is allowed in any handler statements and causes the current exception to be reraised. The exception resumes propagating.
- `FINALLY` contains code that you want executed whether an exception has been raised or not. It should not be used in conjunction with the `CATCH` or `CATCH_ALL` macros.
- `ENDTRY` ends the exception handling block.
- `RAISE (exception_name)` is allowed anywhere and causes a particular exception to start propagating. Valid exception names are any error code you wish to pass to the exception handler. See “Raising Exceptions” below for more details.. (`RAISE` is not shown in the above syntax since it is normally in the `try_block` or in code called by the `try_block`.)

This example shows how an ASLM client can use exception handling:

```
TRY
  DoThisAndThat();  // this function may raise an exception
CATCH(kOutOfMemory)
  prinf("Ran out of memory but continuing on\n");
CATCH_ALL
  printf("Unexpected error. Passing it to next guy up\n");
  RERAISE
ENDTRY
```

In the previous example, if a `kOutofMemory` exception propagates out of the `TRY` block, the first `printf` is executed. If any other exception propagates out of `TRY` block, the second `printf` is executed. In this case, the exception resumes propagating because of the `RERAISE` statement. (If the code is unable to fully recover from the error, or does not understand the error, it needs to further propagate the error to its caller.)

The following is the syntax for using the `FINALLY` macro:

```
TRY
     try_block
[FINALLY
     final_block]
ENDTRY
```

The `final_block` is executed whether or not the `try_block` executes to completion without raising an exception. If an exception is raised in the `try_block`, propagation of the exception is resumed after executing the `final_block`. In other words, if an exception is raised in the `try_block`, it will automatically be reraised after the `final_block` has executed. A `CATCH_ALL` handler and `RERAISE` could be used to do this, but the `final_block` code would then have to be duplicated in two places, as follows:

```
TRY
     try_block
CATCH_ALL
     final_block
     RERAISE;
ENDTRY
{ final_block }
```

A `FINALLY` statement has exactly this meaning, but avoids code duplication.

Note: The behavior of `FINALLY` along with `CATCH` or `CATCH_ALL` clauses is undefined. Do not combine them for the same `try_block`.

Using the exception handling macros

You can use the macros for more than just catching exceptions that are raised by others. For example, if you are entering a section of code that needs to continually check to see whether what you just tried was successful, and if not, to clean up and quit, you can use exception handling to make it easier. Simply put all the code that is “trying to do things” in the `TRY` section and raise an exception by calling `Fail` if anything you try fails. Put your cleanup code in the `CATCH_ALL` or `FINALLY` section depending on whether you want to execute it even if you do not fail.

Raising exceptions

Exceptions are raised by calling either `RAISE`, `Fail`, `FailNULL`, `DebugFail`, or `DebugFailNULL`. The prototypes of these functions are:

```
void RAISE(long errorCode);

void Fail(long errorCode, const char* message);

void FailNULL(void* testValue, long errorCode,
              const char* message);

void DebugFail(long errorCode, const char* message);

void DebugFailNULL(void* testValue, long errorCode,
                   const char* message);
```

The errorCode parameter is the error code that is passed to the exception handler. The exception handler can retrieve the error code by using the ErrorCode macro in the CATCH or CATCH_ALL sections. Likewise, the message parameter is the message string that is passed to the exception handler and can be retrieved with the ErrorMesssage macro. The message parameter can be set to NULL if no message is desired. It defaults to NULL for C++ users.

When calling Fail (or one of its variants), no exception will be raised if the error code kNoError is passed.

When calling Fail (or one of its variants), if a message is passed in the message parameter and you are running the debug version of ASLM (Shared Library Manger Debug), you enter the debugger and the message is displayed.

FailNULL is the same as Fail, except that it only raises an exception if the testValue parameter is NULL.

DebugFail and DebugFailNULL are macros that simply call Fail and FailNULL, except that the message parameter will automatically be set to NULL if qDebug is undefined or is #defined to be 0. This allows you to automatically have the non-debug version of your software omit any message text by simply changing the value of qDebug from 1 to 0.

RAISE is a macro that calls the Fail function and passes NULL for the message parameter. This is the only way to raise an exception that conforms to the DCE standard. All variants of the Fail function are extensions that ASLM has added and are not part of the DCE standard. The ErrorCode and ErrorMessage macros are also extensions added by ASLM.

Rules and conventions for using exceptions

The following rules ensure that exceptions are used in a modular way (so that independent software components can be written without requiring knowledge of each other):

- **Avoid putting code in a try_block that belongs before it.**

  The TRY macro only guards statements for which the statements in the FINALLY, CATCH, or CATCH_ALL clauses are always valid.

  A common misuse of TRY is to put code in the try_block that needs to be placed before TRY. The following example demonstrates this misuse and assumes that open_file will raise an exception if it fails:

```
TRY
     handle = open_file (file_name);
     /* Statements that may raise an exception here */
FINALLY
     close (handle);
ENDTRY
```

The code under FINALLY assumes that no exception is raised by open_file. This is because the code accesses an invalid identifier in the FINALLY section when open_file is modified to raise an exception. The preceding example should be rewritten as follows:

```
handle = open_file (file_name);
TRY
     {
     /* Statements that may raise an exception here */
     }
FINALLY
     close (handle);
ENDTRY
```

The code that opens the file belongs prior to TRY, and the code that closes the file belongs in the FINALLY section. (If open_file raises exceptions, it may need a separate try_block.)

- **Do not place a** `return` **or nonlocal** `goto` **between** `TRY` **and** `ENDTRY`**.**

  It is invalid to return, goto, or leave by any other means a TRY, CATCH, CATCH_ALL, or FINALLY block. Special code is generated by the ENDTRY macro, and it must be executed.

- **Variables that are read or written by exception handling code must be declared volatile.**

  Any variable that is declared outside of the exception handling block, is changed from within the TRY section, and then is referenced later on, should be made volatile by using the Volatile macro. This will prevent the variable from ever being placed in a register. Otherwise you run the risk of having the variable being placed in a register while executing in the TRY section, but after the exception is raised, having the value of the register change.

  Storing local variables in registers is a problem because the TRY macro saves the values of most of the 68000 registers and then when an exception is raised the registers are restored to their saved values. This means that if a variable was stored in a register and was changed in the TRY section, its value will be lost when the exception is raised.

  Generally you do not have to worry about variables that are not referenced frequently, but the only way to be sure that a variable is safe is to look at the compiled code.

Below is an example of how to use the `Volatile` macro:

```
int  temp;
Volatile(temp);
```

Note: The ANSI C `volatile` attribute would normally be used to accomplish this, but is does not work with MPW C++ so the ASLM defines the `Volatile` macro to do the job.

Default exception handlers

When you call `InitLibraryManager`, it installs a default exception handler that catches any exception that is raised and not caught by the client. The default exception handlers are set up and executed by using the C `setjmp`/`longjmp` facility. When you end up in the default exeception handler, your code is executing within the `InitLibraryManager` routine. This does not mean that you entered the default exception handler by calling `InitLibraryManager`. It means that sometime after calling `InitLibraryManager` your application did something that caused an exception to be raised (like trying to call a function in a function set that was not available or could not load), and your application did not set up an exception handler to catch this exception.

The default exception handler will force the application to quit when it catches an exception, and the user will see no warning as to why this happened. If you are running the debug version of the ASLM, you will first end up in the debugger with the message "An exception was thrown and the application did not catch it." When execution continues, the application is forced to quit.

Exceptions and the current client

Exceptions are always passed to (or caught by) the exception handler at the top of the exception handling chain of the current client. Normally, the application that is currently executing is the current client and this is usually the client to which you pass an exception if an exception is raised. If a shared library is made the current client, that shared library must have an exception handler installed if anything is done that can cause an exception to be raised. Unlike clients that call `InitLibraryManager`, shared libraries do not have default exception handlers installed.

Raising an exception when there is no exception handler installed usually results in a crash. When using the debug version of the ASLM, you will first go into the debugger with the message "One too many PopException calls!"

When an exception handler is installed, it is always placed on the exception handling chain of the current client. For this reason it is important that the current client be the same when the exception handler is removed. This means that if you change the current client within the TRY section, you must restore it before entering the FINALLY or ENDTRY sections. When an exception is raised, the current client is automatically restored to the client that was current when the exception handler was installed. Therefore, there is no need to worry about restoring the current client before entering the CATCH or CATCH_ALL sections.

Verifying an object's type

When you create an object, you may want to verify the object's interface (identify its base class) so that you can use the object safely. Otherwise, you cannot safely call any member functions. This is especially true when you use NewObject or are given an object that someone else created. The CastObject and the IsDerivedFrom functions can be used to verify an object's type.

```
void*     CastObject(const void* object,
                     const TClassID parentID, OSErr* err);
void*     CastToMainObject(const void*);
Boolean   IsDerivedFrom(const void*, const TClassID&);
```

The CastObject function casts the object to the specified class and returns a pointer to the object if successful. When using single inheritance, this is always the pointer to the original object. For classes that use multiple inheritance, the object pointer returned may be different.

The object parameter is the object that you want to cast and the parentID parameter is the class ID of the class to which you want to cast the object. If an error occurs, CastObject returns NULL and the OSErr parameter contains the appropriate error code. Possible error codes include kNotFound (when parentID is not a valid class ID) and kNotRelated (when object is not related to parentID).

Any object that you pass to CastObject must have the v-table first. (An object that has its v-table first is an object derived from a base class that has at least one virtual function and no data members. This is true of objects that inherit from the TDynamic, TSimpleDynamic, TStdSimpleDynamic, and TStdDynamic classes.)

CastObject can be called after an object is created to verify its type. For example:

```
// Somebody gave me "theObject", is it really what I asked for?
if (CastObject(theObject, kTMyFirstClassID))
{
      // OK, now it's safe to call theObject methods
      theObject->DoSomethingForMe();
}
```

The IsDerivedFrom function returns true if the object is derived from the specified TClassID. Call IsDerivedFrom only on an object that is implemented in a shared library and is a shared class.

You can use CastToMainObject to obtain the original object pointer without knowing the type of the object. This allows you to get the *real* object when you were given a pointer to one of its multiply-inherited parents.

Verifying a class's base class

Use VerifyClass to verify a class's inheritance. A client can call the VerifyClass function to verify at run time that a given object is derived from a particular base class. If you are using NewObject, you can also do this same verification by passing the required base class type to NewObject. The VerifyClass function allows you to verify a class's interface before actually creating an instance of the class.

```
OSErr VerifyClass(const TClassID, const TClassID parentID);
```

In the following example, VerifyClass verifies that the class with the ID myClassID is derived from the class TParentClass with the ID kMyParentClassID:

```
if (VerifyClass(myClassID, kMyParentClassID))
{
  // now we can use NewObject and safely cast to
  // TMyParentClass
  TMyFirstClass* myObject;
  myObject = (TMyParentClass*) NewObject(myClassID);
  if (myObject != NULL)   // was object created?
  {
     // now it is safe to call methods
     myObject->DoThisAndThat();
  }
}
```

Alternatively, you can call `NewObject` with a second parameter specifying the required base class, as follows:

```
TMyFirstClass* myObject;
myObject = (TMyParentClass*)
     NewObjectWithParent(myClassID, kMyParentClassID);
if (myObject != NULL)  myObject->DoThisAndThat();
```

Using `NewObject`

The global `NewObject` function is a C interface to the `TLibraryManager::NewObject` member function. There are three `NewObject` functions:

```
void* NewObject(const TClassID, OSErr*, TStandardPool*);

void* NewObjectWithParent(const TClassID, const TClassID
                          parentID,OSErr*, TStandardPool*);

void  NewObjectFromStream(const TFormattedStream*,OSErr*,
                          TStandardPool*);
```

`NewObject` creates an object of the class identified by the specified `ClassID`. If `NewObject` cannot find the class, it returns `NULL` and returns an appropriate error code in the `OSErr` parameter. You can pass an optional parent class ID to `NewObject` to verify that the object you are instantiating inherits from the given parent class. The object's `newobject` flag must be set so that the object can be instantiated with a call to `NewObject`. The `newobject` flag for the class is set by the library writer by specifying `flags=newobject` when exporting the class. The only classes shipped with the ASLM that have the `newobject` flag are the `TCollection` and `TScheduler` subclasses. Also, `NewObject` takes an optional `pool` parameter that you can specify if you do not want to allocate memory for the object from the `TLibraryManager`'s object pool.

The `NewObjectWithParent` function works like `NewObject`, but will only create the object if it is a subclass of the parent specified in `parentID`. If it is not a subclass of the parent specified in `parentID`, the error code `kNotRelated` is returned.

The `NewObjectFromStream` function is not supported in version 1.1 of the ASLM.

Loading and unloading the ASLM

Use `UnloadLibraryManager` and `LoadLibraryManager` only for testing purposes. The `IsLibraryManagerLoaded` function checks if the ASLM is loaded. These functions are useful for “resetting” the ASLM, especially if a library remains loaded because a client crashed and you want to get the library unloaded. The Inspector application uses these routines to load and unload the ASLM when requested.

The `LoadLibraryManager` function returns `true` if successful or if the ASLM is already loaded.

```
Boolean   IsLibraryManagerLoaded(void);

Boolean   LoadLibraryManager(void);

void      UnloadLibraryManager(void);
```

Getting the ASLM version

The `GetSLMVersion` function returns the version of the installed ASLM in the `‘vers’` resource format (the first 4 bytes only). If the ASLM extension is not installed, `GetSLMVersion` returns zero.

```
unsigned long GetSLMVersion(void);
```

Sending output to the TraceMonitor window

`Trace` is an I/O method that accepts the same parameters that can be passed to the stdio.h `printf` function in C. It formats unformatted text and sends it to a specified output, usually the TraceMonitor’s Trace window.

```
void  Trace(const char *formatStr, ...);
```

Note: Pascal users can only pass a single parameter to `Trace` (the string to be output).

Entering and leaving system mode

The Macintosh Operating System keeps track of all files opened and closes files used by an application when the application quits. However, an application sometimes makes an operating system call that can cause a file to be opened that should not always be closed when the application quits. An application causing a library file to be opened so that it can be loaded is an example of this. The Macintosh Operating System provides a *system mode* to prevent a file from being closed at the wrong time.

When the Macintosh is in system mode, files that have been opened by an application are not closed when the application terminates. The ASLM goes into system mode when there is a need to open a file that should not be closed when the application quits.

This example illustrates how system mode is used: Assume that you have two applications running at the same time. The first application creates an object that causes a library file to be opened and the library's code to be loaded. From the point of view of the operating system, the application has opened the file.

Now a second application creates an object in the same library. The library file is already open and the necessary code is already loaded, so nothing more needs to be done. If the first application then quits, the operating system ordinarily unloads the library's code and closes the library file. This obviously causes problems for the second application. To avoid this, the ASLM goes into system mode when it opens library files so that they are not closed when the application that is being serviced quits.

You can use `EnterSystemMode` if a library needs to open a file, but wants the file to remain open after the current client quits. In this case, the code to open the file should be preceded by a call to `EnterSystemMode` and followed by a call to `LeaveSystemMode`.

The `EnterSystemMode` function puts the system into system mode. It makes the system heap the current heap, the ASLM the current client, and the ASLM world the current global world. The original heap, current client, and the global world are restored when `LeaveSystemMode` is called. Therefore, every `EnterSystemMode` call must be balanced by a call to `LeaveSystemMode`. You can nest `EnterSystemMode` calls. The `void*` returned by `EnterSystemMode` must be passed to the balancing call to `LeaveSystemMode`.

```
void*     EnterSystemMode();
void      LeaveSystemMode(void*);
```

WARNING Since `EnterSystemMode` changes your global world, model near clients must save their global world by calling `GetCurrentGlobal` before calling `EnterSystemMode` and then restore their global world by calling `SetCurrentGlobalWorld` after calling `EnterSystemMode`. If this is not done, the next call to an exported ASLM routine, including `LeaveSystemMode`, or any call that goes through the jump table, will cause a crash.

Library file and resource management

The ASLM lets you link multiple function sets and classes together into a single shared library, and lets you combine multiple shared libraries into a single file called a library file. Along with the code resources that implement shared libraries, other resources can also be stored in the library file. The ASLM provides a number of routines that are used to open and close library files and also get resources from them in a such a way that they can be shared between multiple clients. Also, routines are provided to add and remove the library file from the resource chain, since the library file is automatically removed from the resource chain once it is opened. This allows users to get resources from the library file by using Macintosh Resource Manager calls.

The library file and resource management functions are declared in the LibraryManagerUtilities.h file. Each routine takes a `TLibraryFile*` parameter that is a pointer to an object that is in charge of the library file. Although it is a C++ object, it can also be retrieved and used by non-C++ users. The `TLibraryFile` object can be retrieved in a number of ways, which are documented in "Getting a Library File's TLibraryFile Object" later in this chapter.

The library file and resource management functions can be used to place the library file's resource fork in the resource chain so Resource Manager calls can work. There are also functions that serve as a front end to certain Resource Manager calls. These functions keep track of the use of resources so clients and libraries can share the resources.

Routines for opening and closing library files and getting the `refNum` for an open library file are also provided.

```
OSErr     Preflight(TLibraryFile*, long* savedRefNum);
OSErr     Postflight(TLibraryFile*, long savedRefNum);

OSErr     OpenLibraryFile(TLibraryFile*);
OSErr     CloseLibraryFile(TLibraryFile*);

TFileSpec*     GetFileSpec(TLibraryFile*);
long           GetRefNum(TLibraryFile*);

Ptr  GetSharedResource(TLibraryFile*, ResType, int theID,
                       OSErr*);
Ptr  GetSharedIndResource(TLibraryFile*, ResType,
                          int index, OSErr*);
Ptr  GetSharedNamedResource(TLibraryFile*, ResType,
                            const char* name, OSErr*);

void      ReleaseSharedResource(TLibraryFile*, Ptr);
long      CountSharedResources(TLibraryFile*, ResType);
```

```
size_t    GetSharedResourceUseCount(TLibraryFile*, Ptr);
OSErr     GetSharedResourceInfo(TLibraryFile*, Ptr,
                                 size_t* theSize,
                                 short* theID, ResType*,
                                 char* theName);
```

Preflight
Postflight

The `Preflight` function places the library file's resource fork in the resource chain so that Resource Manager calls can work. It calls `UseResFile` to make the library file the current resource file and returns the previous current resource file in `savedRefNum`. The `Preflight` function puts the library file just above the system file in the resource chain. (In System 7.1 and later, it is placed just above the System Enabler files.)

Every `Preflight` call must be balanced by a `Postflight` call, which removes the shared library from the resource chain and calls `UseResFile` on the file passed to it. This should be the file returned by `Preflight`.

Both `Preflight` and `Postflight` take long parameters for the `refNum` rather than a short. This is because in future releases, different `TLibraryFile` types may require longs for `refNums`, especially on different platforms.

You must call `Preflight` before you access any of your shared library's resources. You can then make normal Resource Manager calls. You must also call `Preflight` before you make any operating system calls that may try to load a resource from your shared library, such as `GetNewWindow`, `GetMHandle`, and `GetItem`.

If you want to share any resources that you have retrieved with operating system calls, you must keep a use count of them yourself and make sure that they stay loaded and locked until the use count reaches `0`.

You can nest `Preflight` calls. If they are nested, the shared library file is not removed from the resource chain until the outermost `Postflight` is called. However, each nested `Preflight` still causes `UseResFile` to be called for the shared library file, and each nested `Postflight` still causes a `UseResFile` call to be made for the file that was returned by `CurResFile` before the `Preflight` call.

Calling `Preflight` can cause a shared library file to be opened for a client, so it is possible to have the library file opened multiple times, once for each client. This is necessary if a library wants to read in a separate copy of a resource for each client that it has.

If the shared library is loaded, it is already opened, with the ASLM (also called the system client) as the client. If a library calls `EnterSystemMode,` the system client is used by `Preflight`. In this case, any resources that are loaded are shared among all clients. The library must keep track of shared resources itself unless it uses the shared resource calls described later.

If the current client is not the system client, `Preflight` opens the library file on behalf of the client. Any resources loaded are loaded into the current heap zone and cannot be shared with other clients. The library file remains open for the client until the client calls `CleanupLibraryManager`, calls `CloseLibraryFile`, or is unloaded (in the case of a shared library that is a client).

It is important to keep `Preflight` and `Postflight` calls properly balanced. You should not, for example, set up the following situation:

1. Library A calls `Preflight`, and then calls Library B. Library B calls `Preflight` and returns before doing a `Postflight`. Library A calls `Postflight` and then calls back to Library B so Library B can make its `Postflight` call.
2. When library B calls `Postflight`, it calls `UseResFile` on library A, since library A was the current resource file when Library B made its `Preflight` call. This `UseResFile` call fails because Library A is no longer in the resource chain.

Although the above example is not fatal, and may not even cause any problems, it may spark trouble if the client calling the library is relying on the current resource file still being set up properly when the call returns.

Another resource chain problem can arise if a library opens a file for its client after calling `Preflight`. The library must call `UseResFile` on this file after the outermost `Postflight` call if it wants the file to be in the visible resource chain of the client (or the client can do a `UseResFile` on the opened file). Even this does not guarantee that the file will be in the visible resource chain. For example, it will not remain in the chain if the library that opened it was called by another library that had already called `Preflight`. In this situation, it is best to require that the client call `UseResFile` on the opened file.

You do not have to call `Preflight` to get a resource from a client. Clients are already in the resource chain, so `GetResource` calls work as expected. However, if you call `Preflight` first, the client is not in the visible resource chain because `Preflight` calls `UseResFile` on the library file—which, as noticed above, is placed just above the System file and therefore below the client.

In this situation, you must save `CurResFile` before calling `Preflight`. Then, after `Preflight` is called, you must call `UseResFile` to make the client visible again. Note that this always causes the Resource Manager to check all the client files for a resource before checking the library file. This operation may be desirable if you want to allow the client to override resources in the library.

`The Preflight` function acts on the current client's instance of an open library file (see `OpenLibraryFile` below for more information). If the library file is not already opened for the current client, then `OpenLibraryFile` will be called automatically to open the library file for the current client. This means that it is possible to have the library file opened multiple times, once for each client. This is necessary if each client wants its own copy of a resource.

Even if `Preflight` caused the library file to be opened, it will not be closed automatically when `Postflight` is called (See `CloseLibraryFile`, described below, to see when the library file will be closed.).

If the shared library is loaded, its library file is already opened, with the ASLM as the client. If `EnterSystemMode` is called, the ASLM client is used by `Preflight`. In this case, any resources that are loaded will be loaded into the system heap and will be shared among all clients. Users must keep track of shared resources unless they use the shared resource calls described below. If the current client is not the ASLM client, any resources loaded are loaded into the current heap zone and cannot be shared with other clients.

OpenLibraryFile
CloseLibraryFile

The `OpenLibraryFile` and `CloseLibraryFile` functions are used for opening and closing library files.

The `OpenLibraryFile` function allows you to open a library file for the current client. However, the file will not be in the resource chain until you call `Preflight`. The shared library file is opened with read-only access. You can never write a resource to a library file or change a library file resource.

When `OpenLibraryFile` is called, it will open the library file on behalf of the current client. This allows each client of a shared library to have a separate open "instance" of the library file, which allows each client to get resources from the library file that will not be shared. For example, if two different clients call `OpenLibraryFile` on the same library file, the library file will be opened twice, and if each client calls `Get1Resource` on the same resource, they will each get their own copy of the resource.

If `OpenLibraryFile` is called and the library file is already opened for the current client, then all `OpenLibraryFile` does is increment the “open count” for the library file.

The `CloseLibraryFile` function closes the library file for the current client. It only closes the library file if decrementing the open count results in the open count reaching zero. `CloseLibraryFile` can be useful if a client has opened a library file by calling `Preflight` and then `Postflight`, but does not want the library file to remain open until the client calls `CleanupLibraryManager`. In such a case, do not try to close the file by using the `refNum` returned by `GetRefNum`; call `CloseLibraryFile` instead.

When a client quits, `CloseLibraryFile` is called automatically for any library file that was opened for the client. This means that the library file will be closed automatically when a client that is a shared library is unloaded or when a non-library client calls `CleanupLibraryManager`.

GetFileSpec

The `GetFileSpec` function returns `TFileSpec` for the `TLibraryFile`. In version 1.1 of the ASLM, only the `TMacFileSpec` type will be returned. For more information, see “Specifying a Library File” later in this chapter.

GetRefNum

The `GetRefNum` function returns the `refNum` for the open library file for the current client. The `refNum` is cast to a long so that on non-Macintosh systems, it can be a pointer to a structure. Since the `refNum` is a reference to an open file, it lets you perform actions such as reading from the file. You should never attempt to close the file by using the `refNum`.

GetSharedResource
GetSharedIndResource
GetSharedNamedResource

The `GetSharedResource`, `GetSharedIndResource`, and `GetSharedNamedResource` functions keep track of the use of resources so clients and libraries can share the resources. They return a pointer to a shared copy of a specified resource. These calls work just like the Resource Manager’s `Get1Resource`, `Get1IndResource`, and `Get1NamedResource` calls, except that the first time they try to get a given resource

- they call `EnterSystemMode` and `Preflight` before getting a resource
- they call `Postflight` and `LeaveSystemMode` after getting the resource
- they keep track of the resource, so that the next time they try to get the resource, there is nothing more to do but increment the use count and return the pointer to the resource

All of the `GetSharedResource` routines return a pointer to the resource instead of a handle. This is so that there is a chance of getting these routines to port when moving to a system that does not have handles. If you want to obtain a strictly Macintosh resource and do not plan to share it, then you should just call the Macintosh Resource Manager directly after doing a `Preflight`. But if you want to write a portable call to get a resource, or you want to share the resource, you should call one of the `GetSharedResource` functions. Then, instead of treating the result as a resource, you can simply treat it as a pointer to data.

Once you have a pointer, it is possible to call `RecoverHandle` so that you can use your resource to make Memory Manager and Resource Manager calls. However, your code will not be portable and may be unusable by other clients that are sharing the resource—especially if you make Macintosh Toolbox or Operating System calls such as `ReleaseResource`, `DetachResource`, or `HUnlock`.

The `GetSharedResource` routines have `OSErr*` parameters, so you can tell if the routine failed because the resource was not found or because there was not enough memory to read in the resource. You can pass `NULL` for the `OSErr*` parameter if you are not interested in the error.

WARNING `GetSharedNamedResource` takes a C string for the resource name rather than a Pascal string.

The ASLM always locks shared resources using the `HLock` call. Do not unlock shared resources unless you are certain that neither your client nor any other clients depend on the resources being locked.

Note: The shared resource calls will only work if the library file has been opened while the ASLM is in system mode. There are two ways to accomplish this. The first is to simply make sure that a shared library in the library file is currently loaded. This means that it is always safe for code within a shared library to get a shared resource from the shared library's library file. The other way is to first enter system mode by calling `EnterSystemMode` and then call `OpenLibraryFile` to open the library file. See "Entering and leaving system mode" in Chapter 7, "ASLM Utilities" for more details on system mode.

`ReleaseSharedResource`

The ReleaseSharedResource function releases a resource obtained by GetSharedResource, GetSharedIndResource, or GetSharedNamedResource. It decrements the resource's use count. If the use count reaches 0, ReleaseSharedResource calls ReleaseResource to release the resource.

`CountSharedResources`

The `CountSharedResources` function works like `Count1Resource`, except that it calls `EnterSystemMode` and `Preflight` before it calls `Count1Resource`.

`GetSharedResourceInfo`

The `GetSharedResourceInfo` call returns the name, type, size, and flags of a shared resource. You can pass `NULL` to any of the function's parameters if you are not interested in the information it returns.

`GetSharedResourceUseCount`

The `GetSharedResourceUseCount` function returns the use count of a shared resource.

Getting a library's `TLibrary` object

The ASLM provides a number of functions that allow you to obtain a library's `TLibrary` object. The main purpose of obtaining a library's `TLibrary` object is so that a client can call `GetLibraryClientData` to retrieve its own client data for the given library. It is also used for the routines that allow you to explicitly load and unload a library's code segments. See "Per Client Data" in Chapter 7, "ASLM Utilities," for more information on `GetLibraryClientData` and "Support for Explicit Segment Unloading" in Chapter 5, "Writing and Building Shared Libraries," for more information on loading and unloading library code segments.

Although `TLibrary` is a C++ object, it is also useful for non-C++ programmers, since they can still pass the `TLibrary` object pointer into routines such as `GetLibraryClientData`.

```
TLibrary*    GetLocalLibrary();

TLibrary*    LookupLibrary(const TLibraryID);

TLibrary*    LookupLibraryWithClassID(const TClassID);

TLibrary*    LookupLibraryWithFunctionSetID(const TFunctionSetID);

TLibrary*    GetObjectsLocalLibrary(const void* object);
```

A library can call `GetLocalLibrary` to get its own `TLibrary` object. The `LookupLibrary` function returns the `TLibrary` object for the library with the given library ID. The `LookupLibraryWithClassID` function returns the `TLibrary` object for the library that implements the given class ID and `LookupLibraryWithFunctionSetID` returns the `TLibrary` object for the library that implements the given function set

ID. The `GetObjectsLocalLibrary` function returns the `TLibrary` object for the library that implements the given object. See “TDynamic” in Chapter 9, “Utility Classes and Member Functions,” for more details on `GetObjectsLocalLibrary`.

The `GetClassInfo` and `GetFunctionSetInfo` functions can also be used to get the `TLibrary` object for a library. They both provide a way for getting the `TLibrary` object for the function set or class for which you are currently requesting information. See “TClassInfo” in Chapter 9, “Utility Classes and Member Functions,” for more information on `GetClassInfo` and “Getting Information About Function Sets” earlier in this chapter for more information on `GetFunctionSetInfo`.

Getting a library file’s `TLibraryFile` object

The ASLM provides a number of functions that allow you to obtain a library file’s `TLibraryFile` object. The main purpose of obtaining a library file’s `TLibraryFile` object is to make ASLM calls to open the library file and get resources from it. These calls are described in detail in “Library File and Resource Management” earlier in this chapter.

```
TLibraryFile*  GetLocalLibraryFile();
TLibraryFile*  GetLibraryFile(TLibrary*);
TLibraryFile*  GetObjectsLocalLibraryFile(const void*
                                          object);
```

A library can call `GetLocalLibraryFile` to get the `TLibraryFile` object for the library file that the library is in. The `GetLibraryFile` function returns the `TLibraryFile` object for the library file that the library passed to it is in. The `GetObjectsLocalLibraryFile` function returns the `TLibraryFile` for the library that implements the given object. See “TDynamic” in Chapter 9, “Utility Classes and Member Functions,” for more details on the `GetObjectsLocalLibraryFile` function.

The `GetClassInfo` and `GetFunctionSetInfo` functions can also be used to get the `TLibraryFile` object for a library file. They both provide a way of getting the `TLibraryFile` object for the library that implements the function set or class for which you are currently requesting information. See “TClassInfo” in Chapter 9, “Utility Classes and Member Functions,” for more information on `GetClassInfo` and “Getting Information About Function Sets” earlier in this chapter for more information on `GetFunctionSetInfo`.

Per client data

The ASLM provides some support for per client data. This is done by allowing a shared library to maintain a separate data structure for each of its clients. In order for a shared library to have per client data, it must use the `clientdata=` clause in its `Library` declaration in the exports file. This allows the library writer to specify either the name of the structure to be used for per client data or the size of the structure to be used.

The `GetClientData` function is used by the implementation of the library and returns a pointer to the per client data structure for the current client. If this is the first time that this function is called for a given client, the structure is allocated from the client's local pool, and the memory is zeroed. The structure is automatically deallocated when the client terminates or the library is unloaded. *Never* delete this structure. This call should only be made by libraries, and will return `NULL` if called from an application or stand-alone code resource.

The `GetLibraryClientData` function may be used by any client to get its client data for a given shared library. It returns `NULL` if the library does not support client data. See "Getting a Library's TLibrary Object" earlier in this chapter for information on how you can get the `TLibrary` object for a shared library.

```
void*     GetClientData(void);
void*     GetLibraryClientData(TLibrary*);
```

Debugging macros

The `DebugBreak`, `DebugStr`, `DebugTest`, and `DebugBreak` macros are designed to be used while debugging. The `DebugBreak` macro calls `DebugStr` with a specified string. The `DebugTest` macro calls `DebugStr` with a specified string if the `val` parameter is `true`. (`DebugStr`, an A-trap that puts you in the debugger, is documented in *Inside Macintosh*.)

Note: This routine is not available to Pascal users.

Both `DebugTest` and `DebugBreak` generate code only if the variable `qDebug` is defined as 1. (You can define `qDebug` to be 0 or 1 as needed).

With `DebugTest` and `DebugBreak`, you can make a `DebugStr` call that is compiled only when you want debugging on without the inconvenience of having to place `DebugStr` in an `#if` statement each time you want it called. These macros take a C string as a parameter instead of a Pascal string.

```
#define DebugBreak(str)
#define DebugTest(val, str)
```

Using the Global TraceLog

The `GetGlobalTraceLog` and `SetGlobalTraceLog` functions get and set the global `TTraceLog` that belongs to the ASLM.

The `Trace` routine accepts the same parameters that can be passed to the stdio.h `printf` function in C. It formats unformatted text and sends it to a specified output, usually the TraceMonitor's Trace window.

```
TTraceLog*     GetGlobalTraceLog();
void           SetGlobalTraceLog(TTraceLog*);
void           Trace(const char *formatStr, ...);
```

For more information on the TraceMonitor, see "The TraceMonitor Application" in Appendix B.

Specifying a library file

`TFileSpec`

`TFileSpec` is a data structure that is used for specifying the location of a library file (`TLibraryFile`) in a file system or OS independent way. The `TFileSpec` struct is used to compare library files and to pass them around without worrying how the library file is actually specified for the OS or file system being used. The details of the library file's location are stored in a struct that has the `TFileSpec` struct as its first field. This struct is often referred to as a "subclass" of `TFileSpec`. On the Macintosh, the `TMacFileSpec` subclass is used for this specifying the location of library files.

There is also a `TFileSpec` class (and subclasses) for C++ users. C++ users should refer to the "TFileSpec" section of Chapter 9 for details.

Generally you do not need to be concerned with `TFileSpec`s unless you are going to call `RegisterLibraryFile`, `RegisterLibraryFileFolder`, or `GetFileSpec`.

```
typedef unsigned int     FileSpecType;

#define kUnknownType     ((FileSpecType)0)
#define kMacType         ((FileSpecType)1)
#define kMaxType         ((FileSpecType)255)

Boolean IsFileSpecTypeSupported(FileSpecType);
Boolean CompareFileSpecs(const void* f1, const void* f2);
```

```
struct TFileSpec
{
     unsigned char  fType;    /* FileSpec type */
     unsigned char  fSize;    /* size of struct */
};
```

`IsFileSpecTypeSupported` is used to check if the given `FileSpecType` is supported. Generally you will not have a need to use this function.

`CompareFileSpecs` is used to compare two file specs to see if they represent the same file. Note that only a byte compare of the file spec is done. If each file spec represents the same file in different ways, `CompareFileSpecs` will still return `false`.

TMacFileSpec

The `TMacFileSpec` class keeps track of a library file by using a filename, volume refNum, and directory ID. You must use `InitMacFileSpec` to initialize the file spec and make sure that the length is set properly.

```
struct TMacFileSpec
{
     unsigned char  fType;    /* FileSpec type         */
     unsigned char  fSize;    /* size of struct        */
     short          fVRefNum; /* volume refNum of volume
                                 file is on            */
     long           fParID;   /* dirID of the folder file
                                 is in                 */
     Str63          fName;    /* name of the file      */
     };
void InitMacFileSpec(TMacFileSpec *spec, int vRefNum, long
                     parID, Str63 name);
```

Miscellaneous routines

DestroyPointer

The `DestroyPointer` function is used to delete an object when all you know about the object is the `PointerType`. It ensures that if the object is a C++ object, its destructor is called and the proper v-table dispatching is carried out to call the destructor. If the object is not a C++ object, its memory is simply freed. It is used by `TCollection` subclasses to dispose of objects in the collection when `DeleteAll` has been called. You may also find a similar use for it in any routine you write that will destroy objects, and it is left to the user to pass in the `PointerType` of the objects to the routine.

The valid `PointerTypes` are

- `kVoidPointer` for objects that are not C++ objects (so no destructor will be called)
- `kTDynamicPointer` for objects that descend from `SingleObject` and have their v-table first (such as subclasses of `TDynamic` and `TSimpleDynamic`)
- `kTStdDynamicPointer` for objects that do not descend from `SingleObject` and have their v-table first (such as subclasses of `TStdDynamic` and `TStdSimpleDynamic`)
- `kTSCDynamicPointer` for objects that are Symantec C++ objects (such as subclasses of `TSCDynamic`)

Note: `DestroyPointer` does not work for objects that do not have their v-table first.

```
typedef int                  PointerType;


#define kVoidPointer         ((PointerType)0)  /* a non–object pointer */
#define kTDynamicPointer     ((PointerType)1)  /* SingleObject with
                                                  vtable first */
#define kTSCDynamicPointer   ((PointerType)2)  /* a Think C++ object */
#define kTStdDynamicPointer  ((PointerType)3)  /* non-SingleObject with
                                                  vtable first */
void                         DestroyPointer(void*, PointerType);
```

`SLMsprintf`

The `SLMsprintf` function is a special version of the stdio.h `sprintf` function used in C. In code intended to be linked with a shared library, you should use `SLMsprintf` instead of the stdio.h `sprintf` function because `SLMsprintf` is interrupt-safe and because the stdio.h `sprintf` function does not work with shared libraries.

```
int SLMsprintf(char *outString, const char *argp, ...);
```

Note: This function is not available to Pascal users.

Word and byte functions

The `HighWord` function returns the high word of a `long` data type, and `LowWord` returns the low word of a `long` data type.

The `HighByte` function returns the high byte of a word, and `LowByte` returns the low byte of a word.

```
#define HighWord(x)      ((unsigned short)((x) >> 16))
#define LowWord(x)       (((unsigned short)(x)))
```

```
#define HighByte(x)      ((unsigned char)((x) >> 8))
#define LowByte(x)       (((unsigned char)(x)))
```

Memory functions

The `SLMmemcpy`, `SLMmemmove`, and `SLMmemset` functions are equivalent to the C `memcpy`, `memmove`, and `memset` routines. They are exported by ASLM and are faster than the C versions.

```
void  ZeroMem(void* dest, size_t nBytes);
void* SLMmemcpy(void* dest, const void* src, size_t nBytes);
void* SLMmemmove(void* dest, const void* src, size_t nBytes);
void* SLMmemset (void *dest, int c, size_t n);
```

Atomic routines for getting and setting bits

The `AtomicSetBoolean`, `AtomicClearBoolean`, and `AtomicTestBoolean` functions are inline routines that will atomically set, clear, or test a Boolean. A pointer to the boolean is passed as a parameter. The `AtomicSetBoolean` function returns `true` if you were the “setter” and `AtomicClearBoolean` returns `true` if you were the “clearer”. `AtomicTestBoolean` returns the current value of the Boolean.

```
Boolean   AtomicSetBoolean(unsigned char*);
Boolean   AtomicClearBoolean(unsigned char*);
Boolean   AtomicTestBoolean(unsigned char*);
```

The `SetBit`, `ClearBit`, and `TestBit` functions are similar to the atomic boolean routines above, except that they act on bit strings rather than Booleans and `SetBit` and `ClearBit` return the previous value of the bit rather than whether you were the “setter” or “clearer”. The bit string may be any length. The `SetBit` function sets the *n*th bit of a specific block of memory. The `ClearBit` function clears the *n*th bit. Both `SetBit` and `ClearBit` return the value of the bit before it was set or cleared. The `TestBit` function returns the value of the *n*th bit. Each of these routines takes a pointer to the specified block of memory as a parameter. The `bitno` parameter is a zero-based index into the array of bits.

```
Boolean   SetBit(void* mem, size_t bitno);
Boolean   ClearBit(void* mem, size_t bitno);
Boolean   TestBit(const void* mem, size_t bitno);
```

Registering C++ objects with the Inspector

Developers can register C++ objects that they create with the Inspector application so that useful information about the object can be displayed. For each type of object that is registered, the Inspector displays a separate window. The title in the window is the class ID of the object. Each window displays all objects with the given class ID that have been registered. The information for each object is obtained by calling the object's `GetVerboseName` member function. See "TDynamic" in Chapter 9, "Utility Classes and Member Functions," for more information on `GetVerboseName`. See "The Inspector Application" in Appendix B for more information on the Inspector application.

Users register objects with the Inspector by calling `RegisterDynamicObject` and unregister them by calling `UnregisterDynamicObject`. Only subclasses of `TDynamic` may be registered with the Inspector, although you can provide your own base class that forces the v-table first and provides the `GetVerboseName` member function in the same v-table slot as `TDynamic` does. In this case, the object will need to be cast to a `TDynamic*` when it is registered or unregistered.

Objects registered with the Inspector are always added to the beginning of the list in the window. The Inspector updates the contents of any window that has changed each time it gets background or foreground time.

```
void      RegisterDynamicObject(TDynamic*);
void      UnregisterDynamicObject(TDynamic*);
```

8 ASLM Utility Class Categories

This chapter describes the following categories of ASLM utility classes:

- *collection classes* that manage objects organized into lists, arrays, and other kinds of collections
- *object arbitration classes* that handle the sharing of objects among ASLM clients
- *memory management classes* that provide memory pools and other aids to memory management
- *process management classes* that let clients and libraries defer tasks for asynchronous processing
- *miscellaneous classes* that do not belong to any of the other categories and are often used by ASLM clients and by shared libraries

For a complete description of the utility classes that are distributed with the ASLM, see Chapter 9, “Utility Classes and Member Functions.”

Collection classes

The ASLM provides a family of classes that maintain collections of objects. A collection is a data structure such as a linked list or an array, along with a set of routines that can manipulate the collection. The `TCollection` class is the base class for all collections. It provides an interface that lets you use objects in a collection without you having to know any details about the collection.

The `TCollection` class and its subclasses (`TSimpleList, TLinkedList, TPriorityList, TArray, and THashList`) provide access to objects that belong to different kinds of collections. The `TCollection` class and its subclasses also provide member functions for manipulating objects in collections. For example, the `Add` member function adds an object to a collection, and the `Member` member function can tell you if a specified object is in a collection.

Certain `TCollection` member functions such as `AddUnique` and `Member` have versions that take a `TMatchObject` parameter. This parameter gives the collection a user-defined way to compare objects rather than just comparing object pointers, which is what `TCollection` does by default.

When you call `TCollection` member functions that add objects to collections, the data type that you add to collections is `void*`, but you can actually add any data type that fits into `sizeof(void*)` bytes, provided you use a type cast.

The `TIterator` class lets you iterate through all objects in a `TCollection`. You need an iterator when you do not know what kind of data structure is being used for a `TCollection` or you do not have access to the actual data (which should always be the case unless you are implementing a `TCollection` subclass). You can call the `TCollection::CreateIterator` function to create a `TIterator` object for a collection.

Object arbitration classes

Object arbitration is a mechanism for accessing objects by id. It provides functions for registering an object by id and subsequently claiming the object by id for exclusive or shared access. When an object is registered by id—for example, “`ACME:DRAW$RECT`” — then it can be claimed by any ASLM client using this same id provided it has not already been claimed for exclusive access. When the object is registered with an arbitrator, it is attached to a *token*, which is a "carrier" for the object and associates the object with the id. It is this token which is returned when the object is claimed.

Object arbitration is intended to be used to manage access to system or application resources. For example a resource might be a specific physical resource or device driver such as the serial port, or a set of such resources such as all the serial ports on the machine. The owner of a system resource registers an object which provides the interface to the resource and then clients can claim the resource for shared or exclusive access. The choice of shared or exclusive access depends on the service provided by the object and is defined as part of the *access protocol* by the service. This access protocol should be adhered to by clients of the service.

The primary class which provides the arbitration functionality is `TArbitrator`. A `TArbitrator` object is a repository of identified objects that are registered with the arbitrator and are thus available for shared or exclusive access. Any object that has access to a particular instance of the `TArbitrator` class, and can provide the ID of a registered object, can then request an object or can register its own objects by id.

There are several classes that get involved in object arbitration. These classes are described later in this section, but the full descriptions can be found in Chapter 9, “Utility Classes and Member Functions”). The classes used in object arbitration are:

- `TArbitrator`
- `TToken`
- `TRequestToken`
- `TNotifier`
- `TMethodNotifier`
- `TProcNotifier`
- `TTokenNotification`

Registering object with an arbitrator

An object is registered using `TArbitrator::RegisterObject`. The `TArbitrator` creates a `TToken` and stores a pointer to it in an internal hash list. Alternatively a `TToken` can be created first using `TArbitrator::NewToken`, given the object and object id, and then the token can be registered using `TArbitrator:: RegisterToken`.

Once it is registered, the token maintains the following information about the registered object:

- a pointer to the registered object
- the ID under which the object is registered (this ID is used to look up the object)

- a pointer back to the `TArbitrator` object with which the object is registered
- the use count (the number of clients that have claimed the object)
- a `TNotifier` object that can notify the exclusive owner of the token (if there is one) when there is a request from someone else to claim the token

Object ID's (also called token ID's) have to be of a certain format in order to avoid naming conflicts and also in order to group resources (objects) of the same type together. See "Grouping related objects" below and the TToken section in Chapter 9 for more information on the format of an object ID.

Looking up objects and claiming tokens

An *owner* of a token is anyone who has successfully requested a token for either shared access (a *shared* owner) or exclusive access (an *exclusive* owner). When requesting a token, a request type (of type `TokenRequestType,` which is either `kSharedTokenRequest` or `kExclusiveTokenRequest`) is given to request either shared or exclusive access.

This easiest way to lookup an object registered with an arbitrator is to use the `TArbitrator::LookupObject` member function. It returns the actual object that you want to lookup rather than the token. It always does a shared request and simply returns NULL if the request cannot be satisfied.

There are also three functions which can be used to request a token: `GetToken`, `PassiveRequest`, and `ActiveRequest`. The `TArbitrator::GetToken` function requests a token and if the token is available it is returned, otherwise `GetToken` returns `NULL`. The token is *available* if the request type is `kExclusiveTokenRequest` and the token has not been claimed for either exclusive or shared access (the use count is 0) or if the request type is `kSharedTokenRequest` and the token is not already claimed for exclusive access (the use count is >= 0). If the token is claimed for exclusive access then it is not available (the use count is -1).

The other two functions, `TArbitrator::PassiveRequest` and `TArbitrator:: ActiveRequest,` are used to post a request for a token which may not be available. Using these functions, the client can "wait in line" for the token. If you use `ActiveRequest` and the token is not available, the exclusive owner will be notified that there is a request for the object (more on notification later). `PassiveRequest` will not notify the owner. In either case, when the owner releases the token it will then be available to the first requester in line. The request will remain outstanding until it is satisfied or the request token is deleted.

Both PassiveRequest and ActiveRequest return a TRequestToken which is the context for the request. There are two possible states of the request token: either the request succeeded and the token is claimed for the requester, or the request is still pending. A TRequestToken object is a registered token while a request is pending. You can temporarily suspend a request by claiming it exclusively using its Get member function. It remains registered until it is deleted. A TRequestToken is created and returned even if the requested token is not registered. You can call TRequestToken::IsTokenRegistered to check if the token has been registered already. Also, you can force the TArbitrator to create, register, and claim the token by passing true in the registerIfFirst parameter.

There are two ways of waiting for a pending request: polling and notification. To poll for completion of the request you can call TRequestToken::Exchange periodically and if it returns a non-null pointer you are done. The TRequestToken::Exchange member function is used to "trade-in" the request for the real token. Exchange will return NULL if the token is not available, otherwise it will return the real token and delete the request token. Alternatively you can use TRequestToken::GetObject to poll. This will return the token if it is available but will not delete the request token.

If you want to be notified synchronously when the request completes, you can provide a TNotifier when you call PassiveRequest or ActiveRequest. As the owner of a token, if you want to be notified of a pending request then supply the token with a notifier by calling TToken::SetNotifier.

There are two ways an owner can give up ownership. You can call TToken:: Release at any time, or you can call TRequestToken:: Give when you are notified. If you call Release, it will check for an outstanding request and call the Give member function of the request token for you. The requester is notified when the Give member function is called.

Notification

There are two cases where notification is made use of: when an exclusive owner of a token is notified that a request has been made for the token, and when the requester of a token is notified when the request can be satisfied. In both cases, the notification is delivered via a TNotifier object. The token owner sets up his notifier by calling TToken::SetNotifier. The token requester sets up his notifier by passing a TNotifier object to TArbitrator::PassiveRequest or TArbitrator:: ActiveRequest

The TNotifier class is a general-purpose class that provides “object-oriented” callback capability. There are two subclasses of TNotifier provided: TProcNotifier and TMethodNotifier.

When using a TProcNotifier, you provide a notification function of type NotifyProc and optionally a refPtr (as a context pointer). If you use a TMethodNotifier, you provide an object pointer and a member function pointer of type NotifyMethod. The TMethodNotifier calls your member function using your object.

As the owner of a token, when your notification function is called the refPtr is passed as a parameter, and a TTokenNotification is passed as the notifyData parameter. Use TTokenNotification:: GetToken to get the token and TTokenNotification:: GetRequestToken to get the request token. If you want to give up the token then call the request token's Give member function. You should not keep the request token unless you have an agreement with the client as part of your access protocol, and you must not keep the TTokenNotification.

As the requester of a token, when your notification function is called the TTokenNotification is passed as the notifyData parameter. You can get the request token by calling TTokenNotification:: GetRequestToken. When you are notified, the token you requested has already been claimed and is available by calling the Exchange or GetObject member functions of the request token as discussed above.

Grouping related objects

It is possible to manage a set of related objects using the arbitration classes. You can use an object id of the form "<typeID>$<instanceID>". A request can specify only the <typeID>$ portion of the id, in which case the first available object of that type will satisfy the request. In this case it doesn't matter what the <instanceID> portion is of the token that satisfies the request is.

If you have several objects with the same <typeID> (the portion up to the "$"), and more than all of these are claimed exclusively, then a request using ActiveRequest will notify each owner until one gives up the token or they have all been notified. If your access protocol allows owners to keep request tokens, then the first owner that calls TRequestToken::Give will get true back as the result and any subsequent call to Give with the same request token will return false indicating there is no longer an outstanding request.

The owner of a request token may want more than one member of a type, in which case, after the first request is satisfied (you get the first token you requested using TRequestToken::GetObject), you can call TRequestToken::RequestAgain and this will start another active request using the same request token.

Private and global arbitrators

There are two ways to register objects with an arbitrator. One way is to create a private `TArbitrator` object that is recognized only within a specific library or application, or by any other client that knows how to access the arbitrator. The other way is to use the *global arbitrator* that is supplied by the ASLM. This global arbitrator is a `TArbitrator` object that you can retrieve by calling `GetGlobalArbitrator` and that a client or a library can use to register objects for global access.

The `GetGlobalArbitrator` function obtains a global `TArbitrator` object. With a `TArbitrator` object, you can register objects so other clients can look them up.

```
TArbitrator*   GetGlobalArbitrator();
```

A client can obtain the global arbitrator by calling `GetGlobalArbitrator` in the following manner:

```
TArbitrator* arbitrator = GetGlobalArbitrator();
```

An example use of object arbitration

The ASLM does not provide serial port arbitration, but the ASLM object arbitration feature could be used to implement serial port arbitration. As an example of how object arbitration works, suppose an application needs to access a serial port, the ASLM's object arbitration features make it possible to:

- Ask for any available serial port (you might do this if, for example, all serial ports have dial-out modems attached and you don't care which one you get).
- Obtain notification when another client wants to use the serial port that you are using. (Assume, for instance, that you are listening for an incoming call and another application wants to dial out. You can then choose to give up your port.)
- Request a specific serial port.
- Ask for a serial port, even though none is currently available, and receive notification when a serial port becomes available.
- Obtain a group of serial ports (this may be desirable if, for instance, you want to listen for incoming calls on a group of ports dedicated to dial-in modems).

In this example, when you choose your object ids, the `<set id>` could be "`Serport`" and `<member id>` could be "`SLOT0:A`" or "`SLOT0:B`" or "`SLOT1:A`" or "`SLOT1:B`" and so on for multiple serial ports on the main board or on NuBus cards. Then you would have several serial port objects which can be claimed by giving a complete object id to claim a specific port such as "`Serport$SLOT0:A`". Alternatively you might have an application which wants to claim a serial port but doesn't care which one (e.g. they are all connected to outgoing modems). In the latter case you can supply only the set id "`Serport$`" as the object id when making a request and your request will be satisfied by the first serial port available.

More arbitration examples can be found in the ExampleTools folder on the ASLM Examples Disk.

Memory management classes

The ASLM provides memory management classes called *memory pools* that let you allocate memory at interrupt time—an ability that the Macintosh Memory Manager does not have. The ASLM implements memory pools with the `TMemoryPool` class—an abstract base class that provides the interface for all pool classes (see Chapter 9, "Utility Classes and Member Functions" for details). Most of `TMemoryPool`'s member functions are pure virtual member functions that subclasses must override.

Two other classes, `TStandardPool` and `TChunkyPool`, are derived from the `TMemoryPool` class. Both classes support interrupt-safe memory allocation. The `TStandardPool` class lets you allocate variable-size chunks, and `TChunkyPool` allocates only fixed-size chunks, see Chapter 9 "ASLM Utility Classes and Member Functions" for details.

Besides pools that you can create for your own use, there are also several pools created by the ASLM. These include

- the system pool
- the local pool
- the client pool
- the default pool

Sometimes these pools overlap so that the same pool has more than one name. For example, the client pool may be the current client's local pool.

The system pool

The ASLM creates the system pool for use by all ASLM clients. The system pool is allocated out of the system heap and will grow as needed if there is room for it to grow. When you want to allocate memory for system use, you can allocate the memory from the system pool. The system pool can be used for

- objects and memory used by a library that is shared by more than one client
- objects and memory that a shared library keeps for its own private use

When a shared library is loaded, the ASLM automatically sets the library's local pool to the system pool, as explained in the next section, "The Local Pool."

Note: Memory that is allocated for the client's own use—especially if the client is responsible for disposing of the memory—should generally be allocated from the client pool, which is described later in the section, "The Client Pool."

The prototype of the related function is:

```
TStandardPool*      GetSystemPool();
```

The local pool

The local pool is the pool that is attached to the local library manager and is also referred to as the library manager's object pool. When a shared library is loaded, the ASLM installs the system pool as the local pool. When a client calls `InitLibraryManager`, a new memory pool is created for the client's own use and is installed as the client's local pool. The `TLibraryManager` class uses the local pool to allocate memory for classes created with `NewObject` unless you also pass a pool to `NewObject`. There are also other times when a local pool is used for default memory allocations. These situations are described later in "The Client Pool" and "The Default Pool."

You can change the current local pool by calling `SetLocalPool` or by calling `TLibraryManager::SetObjectPool`. This is useful mainly for shared libraries that do not want to use the system pool as their local pool. They can create their own pool and use it as the local pool instead. It can even be shared among a family of libraries.

You should never delete the initial local pool (the one installed by `InitLibraryManager`) since it is deleted automatically when `CleanupLibraryManager` is called. You also should never delete the current local pool, since an attempt to allocate memory from it may be made at a later point. When you have changed the current local pool, you can delete the pool that previously was the local pool as long as it was not the initial local pool. You must make sure that no objects are still allocated from any pool you delete or you will never be able to safely delete those objects.

The prototype of the related functions are:

```
TStandardPool* GetLocalPool();
void           SetLocalPool(TStandardPool*);
```

The client pool

The client pool is the current client's local pool. The current client is normally the currently running application, but also may be set to a shared library or other client by making the proper calls. (Refer to "The Current Client" in Chapter 4, "Writing and Building Clients."

When a shared library needs to allocate memory for a client, the shared library can allocate the memory from the client's local pool rather than from the system pool or from a pool that belongs to the shared library. If a shared library uses the `memory=client` option in its `Library` declaration in its exports file, the shared library allocates memory from the client pool by default.

The `GetClientPool` function returns the client pool.

IMPORTANT The client pool is inaccessible if the current client has not been set up to be an ASLM client. Interrupt and callback routines must make sure that the current client is set up properly before they use the client pool. This is also true of stand-alone code resources that are called from non-ASLM clients.

The prototype of the related function is:

```
TStandardPool*      GetClientPool();
```

The default pool

The default pool is the pool that the ASLM uses for default memory allocations; that is, when the `new` operator is used and a pool is not specified. The default pool is used by the global `new` operator (defined in `GlobalNew.h`) and by the `TDynamic` `new` operator.

The purpose of the default pool is to permit a library to choose whether it wants default memory allocations to come from the library's local pool or the current client's local pool. If the default pool is `NULL`, the current client's pool is used.

The prototype of the related functions are:

```
TStandardPool*      GetDefaultPool();
void                SetDefaultPool(TStandardPool*);
```

The `GetDefaultPool` function gets the default pool and `SetDefaultPool` sets the default pool. For libraries, the default pool is initially set to `NULL` if the library was built with the `memory=client` option; otherwise it is set to the system pool.

If the default pool is set to `NULL` when `GetDefaultPool` is called, `GetDefaultPool` returns the pool that belongs to the current client's local library manager. This is the same as the pool returned by `GetClientPool`. If the default pool is not `NULL` when `GetDefaultPool` is called, the current default pool is returned.

Note: If a library is built using the `memory=client` option and the default pool is then changed to something besides `NULL`, the client pool is no longer used for default memory allocation. In other words, the `memory=client` option causes the initial value of the default pool to be `NULL` only when the library is loaded. The flag has no other effect on memory allocation afterward.

Process management classes

Sometimes clients and libraries must defer tasks for asynchronous processing, possibly for one of the following reasons:

- You need to perform a task that takes a significant amount of time and you want to defer the task for a time when it will be less disruptive.
- You want to do something that involves the operating system, but you are currently executing at interrupt level.
- You want a task to execute after a certain amount of time.
- You want a task to execute while your application is in a certain state (for example, in the foreground or in an event loop).
- You want to accumulate tasks to be executed at the same time for the sake of efficiency.

The ASLM provides two base classes that can be used for asynchronous task processing: `TOperation` and `TScheduler`. A `TOperation` object contains the implementation of a task to be performed. A `TScheduler` object schedules a `TOperation` for later execution and controls when the `TOperation` is executed.

The `TScheduler` class has a number of subclasses that can be used to process operations based on their priority or schedule them to be processed after a certain amount of time has passed, at system task time, or at deferred task time.

The `TScheduler` subclasses are as follows:

- `TTimeScheduler`, which implements a scheduler that processes `TOperation` objects when a requested amount of time has elapsed.
- `TInterruptScheduler`, which is used by interrupt service routines to defer processing.
- `TSerialScheduler`, which ensures FIFO (first in, first out) processing of the tasks.
- `TPriorityScheduler`, which implements a scheduler that lets you serialize tasks by establishing their priorities.
- `TThreadScheduler`, which implements a lightweight “thread” scheduler.
- `TTaskScheduler`, which implements a heavyweight task scheduler.

The most important `TScheduler` member functions are `Schedule`, which schedules a `TOperation`, and `Run`, which processes all scheduled `TOperations`. For more information on the `TOperation` and `TScheduler` classes, and their subclasses and member functions, see Chapter 9, “Utility Classes and Member Functions.”

Miscellaneous classes

The miscellaneous classes provided with the ASLM are the `TDynamic`, `TLibraryManager`, `TClassID`, `TClassInfo`, `TMacSemaphore`, `TTraceLog`, and `TTime` classes. For further details on these classes, consult the alphabetical listings in Chapter 9, “Utility Classes and Member Functions.”

9 Utility Classes and Member Functions

This chapter describes all the ASLM utility classes and their member functions. The declaration of each utility class does not include the private and protected member functions or the data members. Also, the implementation of inline functions is not included. Private and protected functions are used internally by the classes and should not be used by clients.

Class descriptions

The following table shows the inheritance of all the utility classes.

MDynamic					
TAtomicBoolean					
TClassID					
TDynamic	TBitmap				
	TCollection	TArray			
		THashList			
		TSimpleList	TLinkedList TPriorityList		
	TFastRandom	TSimpleRandom			
	THashObject	TArbitrator THashDoubleLong TProcHashObject			
	TIterator	TListIterator THashListIterator TArrayIterator TClassInfo			
	TLibraryFile				
	TLibraryManager				
	TMacSemaphore				
	TMatchObject	TProcMatchObject			
		TToken	TRequestToken		
		TDoubleLong	TTime	TMilleseconds	
				TSeconds	
				TTimeStamp	TStopWatch
				TMicroseconds	
	TMemoryPool	TChunkyPool TStandardPool			
	TNotifier	TProcNotifier TMethodNotifier TPoolNotifier			
	TOperation	TGrowOperation			
	TScheduler	TTimeScheduler			
		TPriorityScheduler	TSerialScheduler TThreadScheduler TTaskScheduler TInterruptScheduler		
	TTestTool				
	TTraceLog				
TFileSpec	TFileIDFileSpec TMacFileSpec				
TFunctionSetID					
TLibraryID					
TLink	TPriorityLink				
TSCDynamic					
TStdDynamic					
TStdSimpleDynamic					
TSimpleDynamic					
TTokenNotification					
TUseCount					

MDynamic

The `MDynamic` class is a base class for shared library classes, which has one virtual function (the destructor). It is meant to be used with multiple inheritance to force the v-table to be at the front of the object when mixing classes. An object that has its v-table first is an object derived from a base class that has at least one virtual function and no data members. For more information on `MDynamic`, see "The TDynamic Family of Base Classes" in Chapter 6, "Using the ASLM."

This class is not a shared class.

Declarations

```
virtual        ~MDynamic();
```

`TArbitrator`

The `TArbitrator` class is used in object arbitration to request and register shared objects.

`TArbitrator` has the following inheritance:

```
TDynamic  -->  THashObject  -->  TArbitrator
```

Description

The `TArbitrator` class is a repository of identified objects that are registered with the arbitrator and are available for shared or exclusive access. An object can access the `TArbitrator` class with the ID of a registered object, and request or register its own objects.

Object arbitration is a mechanism for sharing named objects among ASLM clients. The `TArbitrator` class is a shared data manager, which provides functions for registering and accessing shared data. Programs that make use of shared libraries can share data structures and instances of classes by registering them by name with a `TArbitrator`. The `TArbitrator` class provides facilities for registering data by name, and for requesting shared or exclusive access to the data.

Object arbitration is made possible by an object called a token, which maintains and provides information about objects. A token contains a pointer to the object it represents and the ID that the object was registered with. Tokens are registered with `TArbitrator` objects. The `TArbitrator` class is one of a set of classes that are provided with the ASLM to support object arbitration. The others include `TNotifier`, `TMethodNotifier`, `TProcNotifier`, `TRequestToken`, `TToken`, and `TTokenNotification.`

For more information on `TArbitrator` and object arbitration, see “Object Arbitration Classes” in Chapter 8, “ASLM Utility Class Categories.” The descriptions of the member functions below assume that you have already read this section and understand how object arbitration works. For details on the other classes used in object arbitration, see “TNotifier,” “TMethodNotifier,” “TProcNotifier,” “TRequestToken,” “TToken,” and “TTokenNotification” in this chapter.

Declarations

```
#define kTArbitratorID              "!$arbt,1.1"

#define kRequestIDPrefix '?'
#define kRequestIDPrefixSize 1

typedef int TokenRequestType;

#define kInvalidTokenRequest        ((TokenRequestType)0)
#define kRequestTokenRequest        ((TokenRequestType)1)
#define kExclusiveTokenRequest      ((TokenRequestType)2)
#define kSharedTokenRequest         ((TokenRequestType)3)

                              TArbitrator(TStandardPool* = NULL,
                                          size_t defSize = 0);
virtual                       ~ TArbitrator();

virtual OSErr     RegisterObject(const char* theID, void* theObject);
virtual void*     UnregisterObject(const char* theID);
virtual void*     LookupObject(const char* theID);

virtual OSErr     RegisterToken(TToken*);
virtual TToken*   GetToken(const char* theID, TokenRequestType);

virtual TRequestToken*  PassiveRequest(const char* theID,
                              TokenRequestType, TNotifier* = NULL,
                              BooleanParm registerIfFirst = false);
virtual TRequestToken*  ActiveRequest(const char* theID,
                              TokenRequestType, TNotifier* = NULL,
                              BooleanParm registerIfFirst = false);
virtual TRequestToken*  GetRequest(const char* theID);

virtual Boolean         NotifyOwners(TRequestToken* theRequest);
virtual unsigned long   Hash(const void*) const;

virtual TToken*         NewToken(const char* theID, void* = NULL);
```

Continued on following page ▶

Member functions

`ActiveRequest`

The `ActiveRequest` member function registers a request for a token and notifies the current owner (or owners) that a request is pending. Then `ActiveRequest` returns a `TRequestToken` object. The `ActiveRequest` object works just like `PassiveRequest`, except that the current owner (or group of owners) is notified of the request. More than one owner can be notified if there is more than one token registered with the same type ID. If an owner gives up the token, no more owners are notified. See "Object Arbitration Classes" in Chapter 8, "ASLM Utility Class Categories," for more information on type IDs.

The `TNotifier` parameter is used to provide a notifier that will be called when the requested token becomes available.

If the `registerIfFirst` parameter is `true`, then if the requested token is not already registered, it will be created, registered, and claimed automatically. This solves the race condition problem that will occur if a client wants to register an object only if it is not already registered, but it may be interrupted by another client that wants to do the same thing. If the interrupt comes in after the first client calls `PassiveRequest` (and discovers that the token is not registered already), and before the first client calls `RegisterObject` or `RegisterToken`, then the same object is registered twice.

`GetRequest`

`GetRequest` returns the request token that is being used to handle an outstanding request. The `theID` parameter is used to specify the ID of the token being requested, not the ID of the request token. If there are more than one outstanding requests for the same token, then the first request in line to be satisfied will be returned.

`GetToken`

The `GetToken` member function looks up a token and returns it if it is immediately available. It does not register a request. It returns `NULL` if the token is not available. The `TokenRequestType` parameter is the request type `kExclusiveTokenRequest` or `kSharedTokenRequest`. The exclusive owner of a token can delete the token. This procedure unregisters the token but does not delete the object.

`Hash`

The `Hash` member function obtains the hash value used for storing the `TToken`. It is public in case you want to subclass `TArbitrator` and change the hashing algorithm.

LookUpObject

The `LookupObject` member function returns the object that has been registered with a specified ID. `LookupObject` calls `GetToken` with a request type of `kSharedTokenRequest` and then returns the result of `TToken::GetObject`.

NewToken

`NewToken` is used to create a token that can then be registered by calling `RegisterToken`. The token’s ID and a pointer to the token’s object are passed to `NewToken`.

NotifyOwners

`NotifyOwners` is used to notify owners of a request for a token after `PassiveRequest` has been called and an active request is desired. For example, initially you may only want a token if no one else has claimed it already. However, if at a later point you decide that you would like to request that owners of the token give up the token, then `NotifyOwners` can be called, passing as a parameter the `TRequestToken` returned by `PassiveRequest.`

PassiveRequest

The `PassiveRequest` member function registers a request for a token and returns a `TRequestToken`. It is the same as `ActiveRequest` except that current owners of the token are not notified of the request.

RegisterToken

The `RegisterToken` member function registers a token that was created using `NewToken`. If there is an outstanding request for the token then it will be claimed by the requester before `NewToken` returns. If you wish to claim the token before registering it, call `TToken::Get`.

RegisterObject

You can use `RegisterObject` to register an object without having to deal with tokens. A token is automatically created to hold the object. The `theID` parameter is a string that identifies the object. If there is an outstanding request for the object (actually the token created for the object), then it will be claimed by the requester before `RegisterObject` returns.

Continued on following page ▶

`UnregisterObject`

You can use `UnregisterObject` to unregister a previously registered object. The `theID` parameter is a string that identifies the object. If the object is successfully unregistered, a pointer to the object is returned. Otherwise, `UnregisterObject` returns `NULL`. If the caller plans to delete the object, the caller first needs to make sure that no one is using the object. If this cannot be ensured, the client should instead exclusively claim the object's token first by calling `ActiveRequest` to obtain the token that owns the object, and then delete the token or call `UnregisterToken`.

`UnregisterToken`

You can use `UnregisterToken` to unregister a previously registered token. The normal way to unregister a token is to exclusively claim the token and then delete it. Using `UnregisterToken` allows you to re-use the token.

See also

`TNotifier`
`TMethodNotifier`
`TProcNotifier`
`TRequestToken`
`TToken`
`TTokenNotification`

"Object Arbitration Classes" in Chapter 8, "ASLM Utility Class Categories"

TArbitratorExample1, TArbitratorExample2, and TArbitratorExample3 on the *ASLM Examples* disk

TArray

The TArray class implements an array collection.

The TArray class has the following inheritance:

```
TDynamic  -->  TCollection  -->  TArray
```

Description

The TArray objects can provide efficient and quick indexing into a collection and have the ability to grow as needed. A TArrayIterator class is provided to iterate through the array.

All TArrays are zero-based arrays. Also, there are never any gaps in the array. Removing an object moves all higher-indexed objects down by one index number (for example, if the fifth object in the array is removed, the objects after it all move down one slot to fill in the hole). This also means that you cannot explicitly set or remove the *n*th object in the array. You can think of a TArray as being like a TLinkedList, except it allows you to quickly index objects in the collection.

The TArray constructor's growBy parameter specifies the amount by which a full array should grow. If the number is negative, it represents the percentage by which the array should grow. If the number is positive, it represents the number of cells to add to the array. The initial size of the array is also passed to the constructor, along with the pool that is used to allocate the storage for the array.

Declarations

```
#define kTArrayID "slm:coll$arry,1.1"

               TArray();
               TArray(size_t size, TStandardPool* = NULL,
                      int growBy = 0);
virtual        ~ TArray();

     TStandardPool* GetGrowPool() const;

virtual   TIterator*     CreateIterator(TStandardPool*);

virtual   Boolean        Remove(void*);
virtual   void*          Remove(const TMatchObject&);
virtual   Boolean        Member(const void*);
virtual   void*          Member(const TMatchObject&);

virtual   void*          GetIndexedObject(size_t) const;
```

Continued on following page ▶

Member functions

`CreateIterator`

The `CreateIterator` member function returns a `TArrayIterator` object for the array (see “TArrayIterator” later in this chapter).

`GetGrowPool`

The `GetGrowPool` member function returns the pool that the `TArray` object will use when it needs to grow to support more entries.

`GetIndexedObject`

The `GetIndexedObject` member function is described in “TCollection” later in this chapter.

`Member`

The `Member` member function is described in “TCollection” later in this chapter.

`Remove`

The `Remove` member function is described in “TCollection” later in this chapter.

See also

`TMatchObject`
`TArrayIterator`
`TCollection`

TArrayExample on the *ASLM Examples* disk

TArrayIterator

The `TArrayIterator` class iterates through a `TArray` collection.

The `TArrayIterator` class has the following inheritance:

```
TDynamic  -->  TIterator  -->  TArrayIterator
```

Description

For information on `TArrayIterator`, see “TIterator” later in this chapter.

Declarations

```
#define kTArrayIteratorID "slm:coll$aitr,1.1"

                    TArrayIterator(TArray*);
virtual             ~ TArrayIterator();

virtual   void      Reset();
virtual   void*     Next();

virtual   Boolean   IterationComplete() const;
virtual   Boolean   RemoveCurrentObject();
```

Member functions

IterationComplete

The `IterationComplete` function is described in “TIterator” later in this chapter.

Next

The `Next` function is described in “TIterator” later in this chapter.

RemoveCurrentObject

The `RemoveCurrentObject` function is described in “TIterator” later in this chapter.

Reset

The `Reset` function is described in “TIterator” later in this chapter.

See also

`TIterator`
`TArray`

TArrayExample on the *ASLM Examples* disk

TAtomicBoolean

The `TAtomicBoolean` class atomically sets, clears, and tests a Boolean value.

The `TAtomicBoolean` class has no parent class.

Description

The `TAtomicBoolean` class is simply an inline class to the atomic Boolean routines mentioned in “Atomic Routines for Getting and Setting Bits” in Chapter 7, “ASLM Utilities.” It will set or clear a Boolean and return whether or not you were the setter or clearer. This is all done in an “atomic” matter. In other words, it will work properly even if interrupted by code that tries to set or clear the same Boolean.

Declarations

```
struct TAtomicBoolean
     {
          void                Init();
          Boolean             Set();
          Boolean             Clear();
          Boolean             Test();

          unsigned char       fFlag;
     };
```

Member functions

Init

The `Init` member function is used to initialize the `TAtomicBoolean` and set it to `false`.

Set

The `Set` member function sets the Boolean to `true` and returns `true` if you were the setter of the Boolean.

Clear

The `Clear` member function sets the Boolean to `false` and returns `true` if you were the clearer of the Boolean.

Test

The `Test` member function returns the current value of the Boolean.

TBitMap

The `TBitMap` class is used to store and manipulate a string of bits.

The `TBitMap` class has the following inheritance:

```
TDynamic  -->  TBitMap
```

Description

You can use the `TBitMap` member functions to test, set, and clear the value of specific bits in a block of memory. The member functions are all interrupt safe so no problems arise if you try to set or clear a bit before an interrupt tries to set or clear the same bit. You will always be reliably told the previous value of the bit before you set or cleared it. The bit map array is zero based so the first bit is at index zero.

Both constructors allow you to specify the number of bits in the bit map. The constructor that takes the `pool` parameter allows you to specify the memory pool out of which to allocate the bit map. The other constructor allows you to specify the block of memory to use for the bit map.

Declarations

```
#define kTBitmapID "slm:supp$bmap,1.1"

               TBitmap(size_t numBits, TMemoryPool* pool);
               TBitmap(void* bits, size_t nBits);
virtual        ~ TBitmap();

virtual   Boolean   IsValid() const;

virtual   Boolean   SetBit(size_t);
virtual   Boolean   ClearBit(size_t);
virtual   Boolean   TestBit(size_t);

virtual   long      SetFirstClearBit();
virtual   long      SetFirstClearBit(size_t, size_t);
```

Member functions

ClearBit

The `ClearBit` member function clears the *n*th bit. It returns the value of the bit before it was cleared. The `ClearBit` function does not check to make sure that the index passed to it within range.

IsValid

The `IsValid` member function returns `true` if the `TBitMap` object was initialized properly after it was created. It returns `false` if initialization was not successful. This can happen if there was not enough memory to allocate the block of memory used for the bitmap.

Continued on following page ▶

`SetBit`

The `SetBit` member function sets the *n*th bit of a specific block of memory. It returns the value of the bit before it was set. The `SetBit` function does not check to make sure that the index passed to it within range.

`SetFirstClearBit`

The `SetFirstClearBit` member function sets the first cleared bit. It will return `-1` if there are no cleared bits. Otherwise it returns the index of the bit that was set. The version of `SetFirstClearBit` that takes two `size_t` parameters allows you to specify the range that the bit to set should be in.

`TestBit`

The `TestBit` member function returns the value of the *n*th bit. It does not check to make sure that the index passed to it within range.

TChunkyPool

The `TChunkyPool` class allocates memory of a certain size, called the pool's chunk size.

The `TChunkyPool` class has the following inheritance:

```
TDynamic  -->  TMemoryPool  -->  TChunkyPool
```

Description

The `TChunkyPool` class supports interrupt-safe memory allocation and can be useful when you want to allocate many objects of the same size. One of the more common uses of a `TChunkyPool` is as the link pool for a `TSimpleList`, `TLinkedList`, or `TPriorityList`.

The `TChunkyPool` objects are more efficient than `TStandardPool` objects because they use seven fewer bytes of overhead in each chunk allocated than `TStandardPool` objects. They also increase processing speed because they make it easier to find free chunks in the pool.

The `TChunkyPool` class provides a constant named `kChunkyPoolChunkOverhead` that can help you determine the amount of overhead that each chunk allocated from a pool will require. You should consider the value of this constant when you decide how big a pool you will need.

The definition of kChunkyPoolChunkOverhead is:

```
#define kChunkyPoolChunkOverhead   4
```

The following example shows how to create a `TChunkyPool` object that has enough memory for 200 `TLink` objects:

```
size_t poolsize = 200 * (sizeof(TLink) +
                              kChunkyPoolChunkOverhead);

TChunkyPool* myPool = new (poolsize, kSystemZone)
                              TChunkyPool(sizeof(TLink));
```

The chunk size is always rounded up to a multiple of four, after adding in the required `kChunkyPoolChunkOverhead`. The size of the pool is rounded down to a multiple of the chunk size. Therefore, if you ask for a 100-byte pool with a chunk size of 72, the pool size is 80.

Continued on following page ▶

Declarations

```
#define kChunkyPoolChunkOverhead            4

#define kTChunkyPoolID "!$chkp,1.1"

                         TChunkyPool(size_t chunkSize);
virtual                  TChunkyPool();

virtual   Boolean        IsValid() const;

// TMemoryPool Overrides

virtual   void*          Allocate(size_t size);
virtual   void*          Reallocate(void*, size_t);
virtual   void           Free(void*);
virtual   size_t         GetSize(void*) const;
virtual   Boolean        CheckPool() const;
virtual   size_t         GetLargestBlockSize() const;

          size_t         GetChunkSize() const;
          size_t         GetNumberOfChunks() const;
```

Member functions

Allocate

The `Allocate` member function allocates a block of memory from the pool. When you call `Allocate`, pass the size of the block you want as a parameter.

CheckPool

The `CheckPool` member function returns `true` if no problems are found with the pool. When you are debugging code, it is advisable to call `CheckPool` periodically to make sure that you are not corrupting the pool.

Free

The `Free` member function returns to the pool the block passed to it.

GetChunkSize

The `GetChunkSize` member function returns the pool’s chunk size. When you create a pool, you pass the desired chunk size of the pool to the constructor that creates the pool. This value is passed in the constructor’s `chunkSize` parameter.

GetLargestBlockSize

`GetNumberOfChunks`

The `GetNumberOfChunks` member function returns the number of free chunks available in the pool.

`GetSize`

The `GetSize` member function returns the size of the block passed to it.

`Reallocate`

Memory from a `TChunkyPool` object cannot be reallocated to a different size. Therefore, `Reallocate` returns either `NULL` if a bad memory size is passed (the memory size is greater than the pool's chunk size), or the block of memory passed to it if the size is valid.

`TClassID`

The `TClassID` class represents the class IDs that you use to identify classes implemented in a shared library.

The `TClassID` class has no parent class.

Description

Class IDs are assigned to classes in the library's exports file and are used by clients to specify a class when using routines such as `NewObject` and `LoadClass`.

A `TClassID` object is a simple C string and can be treated as such. When you pass a `TClassID` object to a routine expecting a C string it will be cast to a C string automatically. However, the opposite is not true. C strings must be explicitly cast to a `TClassID` object when needed, such as when calling `NewObject`.

Class IDs take the form xxxx$yyyy. Usually xxxx is related to the developer of the class and yyyy is related to the name of the class.

Adding xxxx ensures that, when combined with yyyy, the class ID will always be unique. Otherwise there would be a lot of classes with a class ID of "TLinkedList" or "TDocument." The xxxx part of the class ID should always start with your four character creator ID, which is assigned by DTS. This is the same creator ID used for applications and documents. Using the creator ID ensures that each developer has a unique ID. You can optionally put something after the creator ID. For example, Apple's DTS group may want to always use "appl:dts" so it only needs to ensure that the yyyy part of the class ID is unique within DTS, but not within all of Apple.

The yyyy part can simply be the class name, such as "TLinkedList," or it can be some sort of abbreviation for the class name, such as "list." The only rule is that when combined with xxxx , it must form a class ID that you know is unique.

Generally your class's class ID will only be found in one place: your library's interface file where a constant of the form `k<classname>ID` is placed. All users of the class ID will just use this constant, including the exports file. In fact, a constant of this form for every class being exported must be made available to the exports file. Since users will usually be using the constant, your class ID does not have to make it clear which class it represents. However, since the class ID appears in the class list of the Inspector, it may be beneficial to give class IDs a descriptive name. This makes debugging easier.

You can (and should) use a version number in your IDs. This allows you to specify a version of a class when you call a function that takes a `TClassID` as a parameter. See Appendix D, "Versioning," for more details on using version numbers. If you use version numbers, your class ID will look something like this:

```
#define kTListID    "appl:dts$TList,1.1"
```

If you are defining many classes with the same version, you may want to do something like this:

```
#define kMyLibaryVersion "1.1"
#define kTListID "appl:dts$TList," kMyLibraryVersion
```

Commas are not allowed in class IDs except at the start of the version number.

Declarations

```
#define kMaxClassIDSize            255

#ifdef __cplusplus

void* operator new(size_t, size_t strLen, TMemoryPool* thePool = NULL)
void* operator new(size_t)
void  operator delete(void* obj, size_t)

      TClassID();
      TClassID(const TClassID&);

      operator const char*() const;       // cast to a const char *

Version     ExtractVersion() const;
size_t      GetLength() const;

TClassID&   operator=(const TClassID&);

Boolean     operator==(const TClassID&) const;
Boolean     operator!=(const TClassID&) const;
```

Continued on following page ▶

There are also global compare operators for comparing a `TClassID` object with a C string and a `TClassID` function for casting a C string to a `TClassID` object.

```
const TClassID& ClassID(const char* str); // cast a char* to a TClassID


Boolean operator==(const TClassID&, const char *);
Boolean operator!=(const TClassID&, const char *);
Boolean operator==(const char *, const TClassID&);
Boolean operator!=(const char *, const TClassID&);
```

If you create a `TClassID` object by invoking the `new` operator (something that you will probably never need to do) you must pass in the size of the class ID string, not including the terminating NULL.

When C++ users pass a C string to a routine expecting a `TClassID` , they must cast it to a `TClassID` first. You can use the `ClassID` function to do this. This example shows how you can perform a cast when you call `NewObject` on a `TLinkedList` :

```
TLinkedList* list = (TLinkedList*)NewObject (ClassID(kTLinkedListID));
```

Member functions

ExtractVersion

The `ExtractVersion` member function extracts version information from the `TClassID` object.

GetLength

The `GetLength` member function obtains the length of a class ID, not including the version information. The maximum size of a `TClassID` is 255 object characters.

operator==
operator!=

The `operator== and operator==` member functions strip off the version numbers when they compare `TClassID` objects. If you want to include the version number when comparing, then use `strcmp`. You should use `strcmp` when comparing `TClassID` objects for ordering purposes (that is, using >, <. >=, and <=).

See also

Appendix D, “Versioning,” for more details on using version numbers in class IDs

`TClassInfo`

The `TClassInfo` class iterates through subclasses of a specified base class, providing information for each subclass.

The `TClassInfo` class has the following inheritance:

```
TDynamic  -->  TIterator  -->  TClassInfo
```

Description

To use the `TClassInfo` class, you must first create an instance of the class by calling the global `GetClassInfo` function and passing it the ID of the base class through which you want to iterate. Each call to `Next` returns a class ID of a class that inherits from the base class. You can call other `TClassInfo` member functions to get information about the class returned by the last call to `Next`. When you are finished with the `TClassInfo` object, delete it in normal C++ fashion.

For more information on `GetClassInfo`, see “TLibraryManager” later in this chapter.

Using `TClassInfo` with function sets

The `TClassInfo` class works with both function sets and classes. To make `TClassInfo` work with a function set that is used by a client, you can give the function set an interface ID by placing the ID in a client’s export file. The interface ID is treated like a class’s parent class ID. This ID (which is entirely fictional, and does not represent a real function set or class) can be used by `TClassInfo` to iterate through all function sets that have the same interface ID. This strategy is useful in conjunction with the `GetFunctionPointer` function. If all function sets with the same interface ID implement the same functions (such as a set of database routines), you can use a `TClassInfo` object to obtain a list of all function sets that implement the desired routines. Then you can let the user choose which one to use.

For more information on interface IDs, see “Getting Information About Function Sets” in Chapter 7, “ASLM Utilities.”

WARNING If the interface ID of a function set conflicts with the `TClassID` of a class or another function set, the function set that is assigned the new interface ID cannot be iterated by a `TClassInfo` object.

Continued on following page ▶

Declarations

```
#define kTClassInfoID "slm:supp$clif,1.1"

virtual                  ~ TClassInfo();

virtual   void           Reset();
virtual   void*          Next();   // safe to cast to
                                   TClassID* or char*

virtual   Boolean   IterationComplete() const;
virtual   Boolean   RemoveCurrentObject();   // do nothing
                                             instead

          void SetBaseClassID(const TClassID& classID);

          TClassID*      GetClassID();
virtual   TClassID*      GetParentID(size_t idx = 0);
          TLibrary*      GetLibrary() const;
          TLibraryFile*  GetLibraryFile() const;
          unsigned short GetVersion() const;
          unsigned short GetMinVersion() const;

          Boolean        GetNewObjectFlag() const;
          Boolean        GetPreloadFlag() const;
          Boolean        GetFunctionSetFlag() const;
          size_t         GetSize() const;
```

Member functions

GetClassID

The `GetClassID` member function returns the `TClassID` object of the class.

GetFunctionSetFlag

The `GetFunctionSetFlag` member function returns `true` if the class is actually a function set.

GetLibrary

The `GetLibrary` member function returns the `TLibrary` object in charge of the library that the class is in.

GetLibraryFile

The `GetLibraryFile` member function returns the `TLibraryFile` object for the library that the class is in.

GetMinVersion

The `GetMinVersion` member function returns the minimum version that the class supports. This value corresponds to the version range you specify when you export a function set or class. When used in conjunction with `GetVersion`, the version range supported by the class can be obtained.

GetNewObjectFlag

The `GetNewObjectFlag` member function returns `true` if the class has its `newobject` flag set.

GetParentID

The `GetParentID` member function returns the parent IDs of the class. Since `GetParentID` works with classes using multiple inheritance, it is necessary to pass in the index of the parent you are interested in. The index is zero-based and defaults to zero. The `GetParentID` function will return `NULL` if the index is out of range, and only returns immediate parents, not parents that are more than one generation away.

GetPreloadFlag

The `GetPreloadFlag` member function returns the `preload` flag. (See "Writing an .exp File" in Chapter 5, "Writing and Building Shared Libraries," for more information on the `preload` flag.)

GetSize

The `GetSize` member function returns the size of the class in bytes. It returns zero if the library in which the class is implemented was not built using the `-sym` option (symbolic debugging symbols enabled).

GetVersion

The `GetVersion` member function returns the version of the class. When used in conjunction with `GetMinVersion`, the version range supported by the class can be obtained.

IterationComplete

The `IterationComplete` member function returns `true` only when `Next` returns `NULL` and the iteration is complete; that is, if the iterator has not become invalid (see "TIterator" later in this chapter for more information on this topic). The iterator can become invalid if `SystemTask` or `GetNextEvent` is called and a shared library is dragged in or out of the Extensions folder, thus adding or removing classes from the system.

Continued on following page ▶

`Next`

The `Next` member function obtains the next subclass, if there is one. The `void*` that is returned may be cast to a `TClassID` or to a `char*`.

`RemoveCurrentObject`

The `RemoveCurrentObject` member function is overridden to do nothing.

`Reset`

The `Reset` member function starts another iteration, beginning with the base class. The base class is the class that was specified when `GetClassInfo` was called to create the `TClassInfo` object, but it can be changed by calling `SetBaseClassID`.

`SetBaseClassID`

The `SetBaseClassID` member function changes the base class through which you are iterating and resets the iterator. This is useful if you have more than one base class through which you want to iterate. If you use `SetBaseClassID`, you do not need to call `GetClassInfo` for each base class.

See also

`TIterator`
`GetClassInfo`

"Getting Information About Function Sets" and "Getting a Library's TLibrary Object" in Chapter 7, "ASLM Utilities"

TClassInfoExample on the *ASLM Examples* disk

TCollection

The `TCollection` class allows you to use objects in a collection without knowing any details about the collection.

The `TCollection` class has the following inheritance:

```
TDynamic  -->  TCollection
```

Description

The `TCollection` class is the base class for all ASLM collection classes. The `TCollection` class and its subclasses (`TSimpleList`, `TLinkedList`, `TPriorityList`, `TArray`, and `THashList`) provide access to objects that belong to different kinds of collections. `TCollection` and its subclasses also provide member functions for manipulating objects in collections. For example, the `Add` member function adds an object to a collection, and the `Member` member function can tell you if a specified object is in a collection.

Most `TCollection` member functions are pure virtual functions, so they must be implemented in subclasses of the `TCollection` class. The `TCollection` classes provided by the ASLM are thread-safe and interrupt-safe, so there is no problem if multiple threads try to change the collection at the same time.

The `TCollection` member functions such as `AddUnique` and `Member` have versions that take a `TMatchObject` parameter. This parameter gives the collection a user-defined way to compare objects rather than just comparing object pointers, which is what `TCollection` does by default.

When you call `TCollection` member functions that add objects to collections, the data type that you add to the collections is `void*`, but you can add any data type that fits into `sizeof(void*)` bytes, provided you use a typecast.

Declarations

```
typedef int          PointerType;

#define kVoidPointer         ((PointerType)0)  /* a non-object
                                                  pointer  */
#define kTDynamicPointer     ((PointerType)1)  /* SingleObject with
                                                  v-table first      */
#define kTSCDynamicPointer   ((PointerType)2)  /* a Think C++ object */
#define kTStdDynamicPointer  ((PointerType)3)  /* non-SingleObject with
                                                  v-table first      */
```

Continued on following page ▶

```
#define kTCollectionID "!$coll,1.1"

virtual                     ~ TCollection();

            size_t          Count() const;
            Boolean         IsEmpty() const;
virtual     TIterator*      CreateIterator(TStandardPool*) = 0;

virtual     OSErr           Add(void*);
virtual     OSErr           AddUnique(void*, const TMatchObject&);
virtual     OSErr           AddUnique(void*);

virtual     void        RemoveAll();
virtual     void        DeleteAll(PointerType = kTDynamicPointer);
virtual     void*       Remove(const TMatchObject&)         = 0;
virtual     void*       Member(const TMatchObject&)         = 0;
virtual     Boolean     Remove(void*)                       = 0;
virtual     Boolean     Member(const void*)                 = 0;

virtual     void*       GetIndexedObject(size_t) const;
            void*       operator[](size_t);

            long        GetSeed() const;
            void        Grab();
            void        Release();
```

Member functions

Add

The Add member function adds to the collection the object that was passed to it. It returns an OSErr. If the object is successfully added to the collection, kNoError is returned in OSErr. If the add does not succeed, an error code is returned in OSErr. The most likely error is kOutOfMemory, although other errors may be possible, depending on the subclass implementation.

AddUnique

The AddUnique member function adds a specified object to the collection if the object is not already in the collection. It returns an OSErr. If the object is successfully added to the collection, kNoError is returned in OSErr. If the add does not succeed, an error code is returned in OSErr. The most likely error is kOutOfMemory, although other errors may be possible, depending on the subclass implementation.

Count

The Count member function returns the number of objects in the collection.

CreateIterator

The CreateIterator member function returns an iterator for the collection (see “TIterator” later in this chapter).

DeleteAll

The DeleteAll member function removes and deletes all objects from the collection. It takes a PointerType parameter that specifies the type of the objects in the collection. Then, if necessary, the objects can be cast to the proper type so the destructors will be called properly. Use kVoidPointer if the objects are not C++ objects (so no destructor will be called). Use kTDynamicPointer for objects that descend from SingleObject and have their v-table first. Use kTStdDynamicPointer for objects that do not descend from SingleObject and have their v-table first. Use kTSCDynamicPointer for objects that are Symantec C++ objects. You cannot call DeleteAll if the collection contains objects that do not have their v-table first. You should instead remove the objects one at a time and delete them yourself. If the collection does not contain pointers—because, for example, you have put long data types in the collection—then you should not call DeleteAll because DeleteAll treats each object as a pointer to memory and attempts to free the memory. Call RemoveAll instead.

If you subclass TCollection, you can use the DestroyPointer function to take care of deleting the pointer for each object in the collection. See “Miscellaneous Routines” in Chapter 7, “ASLM Utilities,” for more information.

WARNING Do not call DeleteAll if the objects in the collection were not allocated using the ASLM global new operator defined in the header file GlobalNew.h. Objects that inherit from TDynamic are always allocated using the ASLM global new operator unless the subclass overrides the new operator. Also, do not call DeleteAll for stack objects or for objects that are defined as data members of a class because these objects are not allocated using the ASLM global new operator. Call RemoveAll instead.

Continued on following page ▶

GetIndexedObject

The `GetIndexedObject` member function returns the *n*th object in the collection. The default implementation of `GetIndexedObject` obtains this information by creating an iterator for the collection and counting as the iterator iterates through the collection until the *n*th object is found. `TCollection` subclasses should override `GetIndexedObject` if there is a more efficient way of getting the *n*th object. The C++ array operator (`operator[]`) simply calls `GetIndexedObject`.

GetSeed

The `GetSeed` member function returns the current seed value (this value changes each time the collection is changed).

Grab

The `Grab` member function grabs the collection's semaphore. It is generally used only by the implementation of `TCollection` and its subclasses.

IsEmpty

The `IsEmpty` member function returns `true` if the collection is empty.

Member

The `Member` member function returns `true` if the object passed to it is in the collection. The `TMatchObject` version of `Member` returns the object that matches the `TMatchObject`. (For more information about the `TMatchObject` class, see "TMatchObject" later in this chapter.)

operator[ ]

The `operator[]` member function calls `GetIndexedObject`.

Release

The `Release` member function releases the collection and semaphore.

Remove

The `Remove` member function removes the object passed to it (or, in the `TMatchObject` version of `Remove`, the object that matches a specified `TMatchObject`). `Remove` removes only the first object that matches the object or `TMatchObject` passed to it. (For more information about the `TMatchObject` class, see "TMatchObject" later in this chapter.)

`RemoveAll`

The `RemoveAll` member function removes all objects from the collection. The user is responsible for making sure that the objects are also deleted if necessary.

See also `TMatchObject`
`TIterator`

TDoubleLong

The `TDoubleLong` class implements a double long (64 bits) integer class that handles all the math functionality of the `TTime` class.

The `TDoubleLong` class has the following inheritance:

```
TDynamic  -->  TMatchObject  -->  TDoubleLong
```

Description

Normally `TDoubleLong` is used as a superclass for some other class which has a 64-bit value as its comparable/hashing value (the default hash value is the low 32-bits). Its main purpose is as the base class for the `TTime` class. It provides all the operators that are commonly used for integer math.

Declarations

```
#define kTDoubleLongID "slm:supp$dbll,1.1"

               TDoubleLong(const TDoubleLong&);
               TDoubleLong(unsigned long low, long hi);
               TDoubleLong(long l);
               TDoubleLong();
virtual        ~ TDoubleLong();

virtual   OSErr          Inflate(TFormattedStream&);
virtual   OSErr          Flatten(TFormattedStream&) const;

virtual   Boolean        IsEqual(const void*) const;
virtual   unsigned long  Hash() const;

virtual   double         ConvertToDouble() const;
                         operator double() const;
                         operator unsigned long() const;
virtual   TDoubleLong&   Add(const TDoubleLong&);
virtual   TDoubleLong&   Subtract(const TDoubleLong&);
virtual   TDoubleLong&   Multiply(const TDoubleLong&);
virtual   TDoubleLong&   Divide(const TDoubleLong&);
virtual   TDoubleLong&   Modulo(const TDoubleLong&);
virtual   TDoubleLong    RShift(unsigned int) const;
virtual   TDoubleLong    LShift(unsigned int) const;
virtual   TDoubleLong&   Negate();
virtual   short          Compare(const void*) const;
```

```
TDoubleLong&   operator=(const TDoubleLong&);
TDoubleLong&   operator+=(const TDoubleLong&);
TDoubleLong&   operator-=(const TDoubleLong&);
TDoubleLong&   operator*=(const TDoubleLong&);
TDoubleLong&   operator/=(const TDoubleLong&);
TDoubleLong&   operator%=(const TDoubleLong&);
TDoubleLong&   operator&=(const TDoubleLong&);
TDoubleLong&   operator|=(const TDoubleLong&);
TDoubleLong&   operator^=(const TDoubleLong&);
TDoubleLong  operator+(const TDoubleLong&) const;
TDoubleLong  operator-(const TDoubleLong&) const;
TDoubleLong  operator*(const TDoubleLong&) const;
TDoubleLong  operator/(const TDoubleLong&) const;
TDoubleLong  operator%(const TDoubleLong&) const;
TDoubleLong  operator&(const TDoubleLong&) const;
TDoubleLong  operator|(const TDoubleLong&) const;
TDoubleLong  operator^(const TDoubleLong&) const;
TDoubleLong  operator~() const;
TDoubleLong  operator-() const;

TDoubleLong  operator<<(unsigned int) const;
TDoubleLong  operator>>(unsigned int) const;

Boolean   operator>(const TDoubleLong&) const;
Boolean   operator<(const TDoubleLong&) const;
Boolean   operator<=(const TDoubleLong&) const;
Boolean   operator>=(const TDoubleLong&) const;
Boolean   operator==(const TDoubleLong&) const;
Boolean   operator!=(const TDoubleLong&) const;
```

Member functions

This section describes the member functions that are not self explanatory.

Compare

The `Compare` member function returns zero if the object passed to it matches the comparison criteria that are specified for `TDoubleLong`. It returns `-1` if the match object is considered to be “greater” and `1` if the object passed to `Compare` is considered to be “greater.” It is normally only used when the `TDoubleLong` object is being used as a `TMatchObject`.

ConvertToDouble

The `ConvertToDouble` member function converts the `TDoubleLong` object to a `double`.

Continued on following page ▶

operator double

This member function performs the same operation as `ConvertToDouble`. It allows for implicit casts to a `double`.

Hash

The `Hash` member function returns the lower 32 bits of the `TDoubleLong`. It is only used when the `TDoubleLong` object is being used with a hash list. It can be overridden by a subclass if a different hash function is needed.

Inflate
Flatten

The `Inflate` and `Flatten` member functions may be useful when streams are supported by future versions of the ASLM.

IsEqual

The `IsEqual` member function returns `true` if the `TDoubleLong` object and the object passed to it are equal. If this not the case, `IsEqual` returns `false`.

RShift
LShift

The `RShift` and `LShift` member functions shift the `TDoubleLong` object the specified number of bits to the left or to the right. The shift is an arithmetic one so there is no rollover.

See also `TMatchObject`

TDynamic

The TDynamic class is a base class that forces the v-table first, overrides new and delete to use memory pools, and provides some non-virtual member functions that provide information about the object.

The TDynamic class has no parent class.

Description

The TDynamic class has some advantages over the TSimpleDynamic class. For example, you can register TDynamic objects with the Inspector and control their tracing. The TDynamic class also provides some member functions that are common in C++ base classes. These member functions (which must be overridden to be useful) include IsValid, which you can use to verify that an object is constructed properly; Clone, which you can use to clone objects; and Inflate and Flatten, which may be useful when streams are supported by future versions of the ASLM.

The main disadvantage of TDynamic is that it has a larger v-table—a wasteful characteristic if you do not take advantage of any of the class's virtual member functions.

For more information on TDynamic, see "The TDynamic Family of Base Classes" in Chapter 6, "Using the ASLM."

Declarations

These are declarations of the TDynamic member functions:

```
typedef int       TraceControlType;

#define kTraceStatus    ((TraceControlType)1)
#define kTraceOn        ((TraceControlType)2)
#define kTraceOff       ((TraceControlType)3)

#define kTDynamicID "!$dyna,1.1"

virtual                 ~ TDynamic;

      void*             operator new(size_t size, TMemoryPool*);
                                          // from specified pool
      void*             operator new(size_t); // from default pool
      void              operator delete(void* obj, size_t)
                                    { SLMDeleteOperator(obj); }
```

Continued on following page ▶

```
const TClassID&         GetObjectsClassID() const;
const TClassID&         GetObjectsParentClassID() const;
       size_t           GetObjectsSize() const;
       TLibrary*        GetObjectsLocalLibrary() const;
       TLibraryFile*    GetObjectsLocalLibraryFile() const;
       TStandardPool*   GetObjectsLocalPool() const;
       void             SetObjectsLocalPool(TStandardPool*) const;

virtual    Boolean     IsValid() const;

virtual    OSErr       Inflate(TFormattedStream&);
virtual    OSErr       Flatten(TFormattedStream&) const;
virtual    TDynamic*   Clone(TStandardPool*) const;

virtual    char*       GetVerboseName(char*) const;
virtual    void        Dump() const;

           void        Trace(char *formatStr, ...) const;
virtual    Boolean     TraceControl(TraceControlType) const;
           Boolean     IsTraceOn() const;
           Boolean     TraceOn() const;
           Boolean     TraceOff() const;

           Boolean     IsDerivedFrom(const TClassID&) const;
```

Member functions

WARNING The following routines may be called only for an object that is implemented in a shared library and is a shared class. If a non-library client implements a class, calling one of these routines on an instance of the class may cause a crash, even if the class inherits from a class that forces the v-table first (even if it inherits from a shared class).

```
IsDerivedFrom
GetObjectsClassID
GetObjectsParentClassID
GetObjectsSize
GetObjectsLocalLibrary
GetObjectsLocalLibraryFile
GetObjectsLocalPool
SetObjectsLocalPool
```

An object that has its v-table first is an object derived from a base class that has at least one virtual function and no data members. This is true of objects that belong to the `TDynamic`, `TSimpleDynamic`, `TStdSimpleDynamic`, and `TStdDynamic` classes.

Clone

The `Clone` member function is used to clone objects. It must be overridden by the subclass to be useful.

Dump

The `Dump` member function displays information about a specified object in the TraceMonitor's Trace window. The `TDynamic` implementation of this member function traces the string returned by `GetVerboseName`. You may want your `TDynamic` subclass to do a formatted trace of all the fields in the object.

Continued on following page ▶

`GetObjectsClassID`

The `GetObjectsClassID` member function returns an object's `TClassID` object. The `TClassID` class is a C string class, so it can be treated as an ordinary C string. If you call `GetObjectsClassID` from a constructor or a destructor, `GetObjectsClassID` returns the `TClassID` object for the class whose constructor or destructor calls `GetObjectsClassID`, not for the subclass. For example, if `TSquare` inherits from `TShape` and the constructor for `TShape` calls `GetObjectsClassID`, then when you create a `TSquare` object, the call by the constructor for `TShape` returns the `TShape` class ID, and the call by the constructor for `TSquare` returns the `TSquare` class ID.

`GetObjectsLocalLibrary`

The `GetObjectsLocalLibrary` member function returns the `TLibrary` object for the library in which the object is implemented. If you call `GetObjectsLocalLibrary` to obtain information for a polymorphic object, the member function always returns `TLibrary` for the subclass, not `TLibrary` for the base class. For example, if your object is a `TSquare` subclass, but all you know is that it has `TShape` as a base class, `GetObjectsLocalLibrary` returns the `TLibrary` object for `TSquare`, not the `TLibrary` object for `TShape`. If you are in a constructor or destructor when you call `GetObjectsLocalLibrary`, the member function returns `TLibrary` for the library of the class whose constructor or destructor you are in, not `TLibrary` for the subclass. This behavior is similar to that of `GetObjectsClassID`, above.

`GetObjectsLocalLibraryFile`

The `GetObjectsLocalLibraryFile` member function returns the `TLibraryFile` object for the library in which the object is implemented. If you call `GetObjectsLocalLibraryFile` to obtain information for a polymorphic object, the member function always returns the `TLibraryFile` for the subclass, not the `TLibraryFile` for the base class. This is similar to the behavior of `GetObjectLocalLibrary`. If you are in a constructor or destructor when you call `GetObjectsLocalLibrary`, the member function returns `TLibraryFile` for the library of the class whose constructor or destructor you are in, not `TLibraryFile` for the subclass. This behavior is similar to that of `GetObjectsClassID.`

GetObjectsLocalPool

The `GetObjectsLocalPool` member function is similar to `GetObjectsLocalLibraryFile` except that it returns the local pool for the object's shared library. If you are in a constructor or destructor when you call `GetObjectsLocalPool`, `GetObjectsLocalPool` returns the local pool for the library of the class whose constructor or destructor you are in, not the local pool for the subclass. This behavior is similar to that of `GetObjectsClassID`.

GetObjectsParentClassID

The `GetObjectsParentClassID` member function returns the `TClassID` object for the parent class of the object. If you are executing a constructor or destructor when you call `GetObjectsParentClassID`, what you obtain is `TClassID` for the parent class of the class whose constructor or destructor you are in, not the subclass. This behavior is similar to that of `GetObjectsClassID`.

GetObjectsSize

The `GetObjectsSize` member function returns the size of the object in bytes. It returns zero if the library in which the class is implemented was not built using the `-sym` option (symbolic debugging symbols enabled).

GetVerboseName

The `GetVerboseName` member function returns a string that describes the object. You must pass a pointer to 256 bytes of memory as a parameter to `GetVerboseName`. The member function returns information about the object in that 256-byte parameter. (The `TDynamic` implementation of the member function returns a string containing the address of the object.) If you override `GetVerboseName`, make sure that the information which `GetVerboseName` generates fits on one line and is 256 characters or less in length, including the terminating NULL character.

Inflate
Flatten

The `Inflate` and `Flatten` member functions may be useful when streams are supported by future versions of the ASLM.

IsDerivedFrom

The `IsDerivedFrom` member function returns `true` if the object is derived from the specified `TClassID` object.

Continued on following page ▶

IsValid

The `IsValid` member function returns `true` if the object was initialized properly after it was created. Some classes always initialize properly, but others may need to allocate memory or get resources during construction. Classes whose construction can fail must override `IsValid` to return `false` if construction fails; otherwise, `IsValid` always returns `true`.

SetObjectsLocalPool

The `SetObjectsLocalPool` member function sets the local pool for the object's shared library. For more information see "Memory Management Classes" in Chapter 8, "ASLM Utility Class Categories."

Trace

The `Trace` member function sends output to the TraceMonitor's Trace window. The `Trace` function, like `printf`, takes an unformatted string with multiple parameters. For more information see "Sending Output to the TraceMonitor Window" in Chapter 7, "ASLM Utilities."

TraceControl

The `TraceControl` member function turns an object's tracing on and off. It takes only one parameter: a constant that specifies what the member function should do. If the `kTraceStatus` constant is specified as a parameter, `TraceControl` returns `true` if tracing is on and returns `false` if tracing is off. The constant `kTraceOn` turns tracing on and returns the tracing state before it was turned on. The constant `kTraceOff` turns tracing off and returns the tracing state to the state it was in before it was turned off. The `TraceControl` function was created so that only one virtual function (instead of three) would be needed to handle `IsTraceOn`, `TraceOn`, and `TraceOff`.

IsTraceOn
TraceOn
TraceOff

The `IsTraceOn`, `TraceOn`, and `TraceOff` member functions call `TraceControl` with the appropriate argument. The `IsTraceOn` function returns `true` if tracing is on for the object. The `TraceOn` and `TraceOff` functions turn tracing on and off for the object. Note that the `TDynamic` implementation of `TraceControl` does not support turning tracing on and off. In the `TDynamic` implementation, tracing is always on. The `TDynamic` subclass must override `TraceControl` to turn off tracing for the object. The `TDynamic::IsTraceOn` function always returns `true`. You may want to override `TraceControl` in your `TDynamic` subclass and maintain a trace flag.

TFastRandom

The TFastRandom class returns a random number computed with 32-bit arithmetic.

The TFastRandom class has the following inheritance:

```
TDynamic  -->  TFastRandom
```

Description

The TFastRandom class creates random numbers according to the following algorithm (using the current time stamp as the initial seed):

```
Seed = (Seed*2416 + 374441) % 1771875.
```

Declarations

```
#define kTFastRandomID "slm:supp$frnd,1.1"

const unsigned long kMaxFastRandom = 1771874;

                         TFastRandom();
                         TFastRandom(unsigned long seed);
virtual                  TFastRandom();

virtual   void           SetSeed(unsigned long seed);
virtual   void           SetSeed();
          unsigned long  GetSeed() const;

virtual   unsigned long  GetRandom();
virtual   unsigned long  GetRandomNumber(unsigned long lo,
                                         unsigned long hi);
```

Member functions

TFastRandom

The TFastRandom member function creates an object, using the current time stamp as the seed. TFastRandom(unsigned long *seed*) creates the object using *seed* as the seed.

GetRandom

The GetRandom member function returns a random number ranging from 0 to kMaxFastRandom, inclusive. You should not normally use this member function; instead, use GetRandomNumber.

GetRandomNumber

The GetRandomNumber member function returns the a random number ranging from lo to hi, inclusive.

Continued on following page ▶

GetSeed

The `GetSeed` member function returns the current seed value. The seed value changes each time `GetRandom` is called.

SetSeed

The `SetSeed` member function sets the random number seed using the current time stamp.

TFileSpec

The TFileSpec class is a base class for specifying the location of a library file (a TLibraryFile object) in a file system-independent or OS-independent way.

The TFileSpec class has no parent class.

Description

The subclasses of TFileSpec contain the details of a library file's location. The TFileSpec base class is used to compare TLibraryFile objects to see if they represent the same file and also so a file specification can be passed around without worrying about the contents.

The TFileSpec class has two subclasses: TMacFileSpec and TFileIDFileSpec. The TMacFileSpec class keeps track of files by volume refNum, directory ID, and filename. The TFileIDFileSpec class keeps track of files by volume refNum and file ID.

There are also C struct and Pascal Record versions of TFileSpec and its subclasses for C and Pascal users.

In version 1.1 of the ASLM, only the TMacFileSpec subclass is supported, since the ASLM currently uses the TMacFileSpec subclass to keep track of library files on the Macintosh Operating System.

The TFileSpec class provides cast operators to automatically cast a TFileSpec object to TMacFileSpec or TFileIDFileSpec. It is your responsibility to ensure that this cast is a legal one. You can call TFileSpec::GetType to get the type of the TFileSpec object.

Generally, you do not have to be concerned with TFileSpecs unless you plan to call RegisterLibraryFile, RegisterLibraryFileFolder, or GetFileSpec.

Declarations

```
typedef unsigned int FileSpecType;

#define kUnknownFileSpecType        ((FileSpecType)0)
#define kMacType                    ((FileSpecType)1)
#define kFileIDType                 ((FileSpecType)2)
#define kMaxFileSpecType            ((FileSpecType)255)

class TFileSpec;
class TMacFileSpec;
class TFileIDFileSpec;

extern "C" Boolean IsFileSpecTypeSupported(FileSpecType);
extern "C" Boolean CompareFileSpecs(const void* f1, const void* f2);
```

Continued on following page ▶

```
void*             operator new(size_t size, TMemoryPool *thePool)
void*             operator new(size_t size)
void              operator delete(void* obj, size_t)

FileSpecType      GetType() const;
unsigned char     GetSize() const;

// compare operators

Boolean           operator==(const TFileSpec&) const;
Boolean           operator!=(const TFileSpec&) const;

// cast operators

                  operator const TMacFileSpec&() const;
                  operator const TFileIDFileSpec&() const;

unsigned char     fType;
unsigned char     fSize;
```

Member functions

`operator == and operator !=`

The `operator` member functions can be used to compare two file specifications. It does not matter whether the two file specs are of the same subclass.

`GetSize`

The `GetSize` member function returns the size of the `TFileSpec` data structure.

`GetType`

The `GetType` member function returns the type of the `TFileSpec` data structure.

See also

`TFileIDFileSpec`
`TMacFileSpec`

TFileIDFileSpec

The `TFileIDFileSpec` class is a `TFileSpec` subclass that keeps track of library files by file ID and volume refNum.

The `TFileIDFileSpec` class has the following inheritance:

```
TFileSpec  -->  TFileIDFileSpec
```

Description

In version 1.1, the ASLM uses the `TMacFileSpec` subclass to keep track of library files on the Macintosh Operating System.

You can use the `TFileIDFileSpec` for your own purposes if you wish, but do not pass them to any ASLM routines. See “TFileSpec” for more information.

Declarations

```
// Some macros to make accessing fields without doing a cast easier

#define GetFileIDFromFileSpec(x)  (((const TFileIDFileSpec&)x).fFileID)
#define GetVRefNumFromFileSpec(x) (((const TFileIDFileSpec&)x).fVRefNum)

extern "C" void InitFileIDFileSpec(TFileIDFileSpec *spec, int vRefNum,
                                    long fileID);

      TFileIDFileSpec(const TFileIDFileSpec&);
      TFileIDFileSpec(int vRefNum, long fileID);

short fVRefNum;              // volume refNum
long  fFileID;               // FileID
```

See also

`TFileSpec`

TFormattedStream

`TFormattedStream` is not yet implemented.

TFunctionSetID

The `TFunctionSetID` class is a class that you can use to identify function sets implemented in a shared library.

The `TFunctionSetID` class has no parent class.

Description

A `TFunctionSetID` object, like a `TClassID` object, is a C string made up of a developer ID and a class name separated by a dollar sign ($), and optionally followed by version information.

Function set IDs are assigned to function sets in the library's exports file and are used by clients to specify a function set when using routines such as `GetFunctionPointer` and `LoadFunctionSet`. For C users, there is also a `TFunctionSetID` typedef.

Function-set IDs are written in this format:

```
xxxx$MyFunctionSet[,1.2.3]
```

Although `TFunctionSetID` objects work exactly like `TClassID` objects (the two can be used interchangeably), `TFunctionSetID` makes it clearer what parameters are expected for certain routines. For example, function set users can call `LoadFunctionSet(TFunctionSetID*)` instead of `LoadClass(TClassID*)`, but either can be used to perform the task of loading a function set.

The `FunctionSetID` functions perform casts in a way similar to the `ClassID` function. It is only used by (and required by) C++ users. For example, you can call `LoadFunctionSet` using the following format:

```
OSErr err = LoadFunctionSet FunctionSetID(kMyFunctionSetID));
```

See "TClassID" earlier in this chapter for details on `TFunctionSetID`s. All the information provided for `TClassID` objects is also true of `TFunctionSetID` objects, but keep in mind that `TClassID` member functions are meaningless to C users.

Declarations

`TFunctionSetID` is typedef'd to be the same as `TClassID`.

```
#define   TFunctionSetID TClassID
```

See also

`TClassID`

TGrowOperation

TGrowOperation objects are used to automatically increase a pool's size when the pool comes dangerously close to running out of memory.

The TGrowOperation class has the following inheritance:

```
TDynamic  -->  TOperation  -->  TGrowOperation
```

Description

This class is used automatically by TPoolNotifier when it needs to increase the pool size at interrupt time, in which case it schedules a TGrowOperation on the global TTaskScheduler.

The TGrowOperation is not processed at interrupt time so it is always safe for it to grow the pool.

This class is only used by the ASLM; you will never need to use this class in your programs.

Declarations

```
#define kTGrowOperationID         "!$gwop,1.1"

                                  TGrowOperation();
virtual                           ~ TGrowOperation();

virtual void                      Process();

size_t                            fGrowBy;
Boolean*                          fOpInUse;
```

THashDoubleLong

The THashDoubleLong class is used to hash a TDoubleLong.

The THashDoubleLong class has the following inheritance:

```
TDynamic  -->  THashObject  -->  THashDoubleLong
```

Description

The THashDoubleLong class should be subclassed to provide a useful Hash member function. It has no use in ASLM version 1.1. However, if subclassed, it can be used as a Hash object for a TDoubleHashList if you write one.

Declarations

```
#define kTHashDoubleLongID "slm:supp$hdbl,1.1"


                              THashDoubleLong();
virtual                       ~ THashDoubleLong();

virtual unsigned long         Hash(const void*) const;
```

Member functions

Hash

The Hash member function returns the pointer to the object that is passed to it. It should never be overridden by a subclass. The const void* parameter is a pointer to a TDoubleLong object.

`THashList`

The `THashList` class implements a hash list as a `TCollection` subclass.

The `THashList` class has the following inheritance:

```
TDynamic  -->  TCollection  -->  THashList
```

Description

The hash list implemented by `THashList` is a chained hash list. The hash list has an array of buckets. When only one object hashes to a particular bucket, then the bucket contains a pointer to the object. If more than one object hashes to the same bucket then the bucket will contain a pointer to a linked list of objects that hash to that bucket.

The `THashList` class implements all of the standard `TCollection` member functions, plus the extra member functions listed in the declaration below.

Using a hash list

To make a hash list useful, you must create a subclass of `TMatchObject` that is aware of the kinds of objects added to the hash list. When you call member functions that use a `TMatchObject`, such as `Member(TMatchObject&)`, the `Hash` member function of the match object is called to get the hash key for each object.

Each object in the hash list that also has the same hash key is passed to the match objects' `Compare` member function to determine whether it matches the match object. For example, if a hash list contains objects that are hashed by a name contained in the object (such as a person's name), the match object should also contain a name field that it can match with the name. Also, the object's `Hash` member function should return the hash value of that name using the same hashing function that is used by the `THashObject` belonging to the `THashList`.

To make things easier, you can use the `TProcHashObject` and `TProcMatchObject` classes and set their `HashProc` to the same function.

Declarations

```
struct HashListInfo
{
  size_t  emptySlots;    // number of empty slots in
                         // the hash list
  size_t  singleSlots;   // number of slots with only
                         // one entry
  size_t  numChains;     // number of slots with more
                         // than one entry
  size_t  longestChain;  // the longest chain
  size_t  avgChain;      // the average length of a chain
};

#define kTHashListID     "!$hsls,1.1"

                         THashList(BooleanParm);

                         THashList();
                         THashList(THashObject*,
                                   size_t initialSize,
                                   TMemoryPool* = NULL);
virtual                  ~ THashList();

virtual   OSErr          Grow(size_t newSize);
          OSErr          Rehash();

          void           SetHashObject(THashObject*);
          TMemoryPool*   GetLinkPool() const;
          THashObject*   GetHashObject() const;
          size_t         GetTableSize() const;
          void           SetLinkPool(TMemoryPool*);

virtual   void        GetHashListInfo(HashListInfo&) const;

// TCollection Overrides

virtual   TIterator*  CreateIterator(TStandardPool*);

virtual   void*       Remove(const TMatchObject&);
virtual   void*       Member(const TMatchObject&);
virtual   Boolean     Remove(void*);
virtual   Boolean     Member(const void*);
```

Continued on following page ▶

Member functions

CreateIterator

The `CreateIterator` member function returns an iterator for the collection (see “TIterator” later in this chapter).

GetHashListInfo

The `GetHashListInfo` member function returns information about the hash list in a `HashListInfo` data structure. The `HashListInfo` data structure includes the following fields:

- `emptySlots` is the number of empty hash buckets in the hash table.
- `singleSlots` is the number of hash buckets with only one entry.
- `numChains` is the number of hash buckets with more than one entry, thus requiring the hash bucket to point to a list of entries rather than pointing directly to the entry.
- `longestChain` is the longest chain hanging off a hash bucket, in other words, the most entries that hash to the same hash bucket.
- `avgChain` is the average length of a chain. Hash buckets with 0 or 1 entries are not included in this average.

Use this information to determine if your hash list is big enough and if you are using a good hash function. If the proportion of empty hash buckets to hash buckets that require chaining (especially long chains) is high, then your hash function is not very good. The ideal hash function would result in each hash bucket getting one entry before chaining is started (an impossible task with random data). The worst hash function would hash all entries into the same hash bucket.

GetHashObject

The `GetHashObject` member function returns the `THashObject` being used by the `THashList`.

GetLinkPool

The `GetLinkPool` member function returns the pool in which `TLink` objects will be allocated when chaining is required.

GetTableSize

The `GetTableSize` member function returns the number of hash buckets in the `THashList` object.

Grow

The `Grow` member function changes the number of buckets in the `THashList`. This member function can be called only at System Task time, unless you have already forced the `THashListGrower` class to be loaded.

The `Grow` function is interrupt-safe, but not reentrant. The error code `kNotAllowedNow` is returned if the `THashList` is already being grown. The other possible error return value is `kOutOfMemory`.

Member

The `Member` function is described in “TCollection” earlier in this chapter.

Rehash

The `Rehash` member function forces the `THashList` to rehash itself. This is not necessary in the current implementation of `THashList`, but was required in an earlier version of the ASLM by a class named `TDoubleHashList`. If the `TDoubleHashList` class is ever reactivated, it will require this member function. Error return values and caveats for `Rehash` are the same as for `Grow`.

Remove

The `Remove` function is described in “TCollection” earlier in this chapter.

SetHashObject

The `SetHashObject` member function sets the `THashObject` that the `THashList` should use for hashing.

SetLinkPool

The `SetLinkPool` member function sets the pool that will be used when allocating links for chaining is required. By default, the pool passed to the constructor is used as the initial link pool.

THashList

The empty `THashList` constructor uses an `initialSize` of 103, the current client’s pool, and does not set the hash object. Its hash function simply takes the address of the object to be hashed.

The second `THashList` constructor creates a hash list whose hash table is the size specified in `initialSize`. The constructor allocates the hash table from the specified pool, and allocates the links for the chains from the same pool. The `THashObject` passed to the constructor is used to do the hashing.

See also

`THashObject`
`TMatchObject`
`TCollection`
`THashListIterator`

THashListIterator

The THashListIterator class iterates a THashList collection and most subclasses of a THashList.

The THashListIterator class has the following inheritance:

```
TDynamic  -->  TIterator  -->  THashListIterator
```

Description

The THashListIterator interface is the same as all other iterators. A TMatchObject may be used if you only want to look at objects in a single hash bucket. The TMatchObject must have overridden the Hash member function, and must use the same hashing algorithm used by the THashObject given to the THashList. The overridden Hash member function is only called once by the iterator to get the hash value of the match object. Only objects in the hash list that hash to this same value and are considered by the match object to be equal to the match object are returned by the iterator.

For more information, see “TIterator,” “TMatchObject,” and “THashList” in this chapter.

Declarations

```
#define kTHashListIteratorID "!$hsit,1.1"


               THashListIterator(THashList*,
                                   TMatchObject* = NULL);
virtual        ~ THashListIterator();

// TIterator Overrides

virtual   void           Reset();
virtual   void*          Next();

virtual   Boolean        IterationComplete() const;
virtual   Boolean        RemoveCurrentObject();
```

Member functions

Next

The Next member function returns the next object in the hash list. If a match object was given to the iterator then only objects that match the match object and hash to the same bucket as the match object will be returned. See “TIterator” in this chapter for more information.

IterationComplete

The `IterationComplete` member function returns `true` if the iteration completed successfully. See “TIterator” in this chapter for more information.

RemoveCurrentObject

The `RemoveCurrentObject` member function removes the current object (the one just returned by `Next`) from the hash list. See “TIterator” in this chapter for more information.

Reset

The `Reset` member function will reset the iterator so the entire hash list can be iterated over again. See “TIterator” in this chapter for more information.

See also

`TIterator`
`THashList`
`TMatchObject`

THashObject

The THashObject class is the base class for all objects that “know” how to hash another object.

The THashObject class has the following inheritance:

```
TDynamic  -->  THashObject
```

Description

The THashObject class is normally subclassed to provide a hashing function for a particular type of data or class. The most common use of a THashObject is with the THashList which requires the user to pass it a THashObject that it will use to get a hash value for objects added to the hash list. In this case, the THashObject subclass’s Hash member function must know what type of objects are being added to the hash list so it can cast the object past to it to the proper type and calculate the hash value for the object. Normally this involves looking at a field of the object and calculating the hash value based on what is in the field. For example, a class called TPersonRecord might have an fName field that is simply a C string. The Hash member function could simply return the sum of the ASCII characters as the hash value.

Declarations

```
virtual                  ~ THashObject();

virtual   unsigned long  Hash(const void*) const  = 0;
```

Member functions

Hash

The Hash member function will return the hash value for the object passed to it. The Hash member function must know what type of object is passed to it so it can cast it properly.

See also

THashList

TInterruptScheduler

The TInterruptScheduler class is used by interrupt service routines to defer processing.

The TInterruptScheduler class has the following inheritance:

```
TDynamic --> TScheduler --> TPriorityScheduler --> TInterruptScheduler
```

Description

On the Macintosh, TInterruptScheduler is a front end to the Deferred Task Manager. Operations scheduled on the TInterruptScheduler execute at deferred task time.

The TInterruptScheduler cannot be used on a MacPlus when System 6 is running. The problem is that the Deferred Task Manager does not exist in this situation, and since the TInterruptScheduler is just a front end to the Deferred Task Manager, it cannot operate without it. In this case the Schedule member function will not do anything and IsValid will return false.

The TInterruptScheduler provides an alternate constructor that takes another TScheduler as a parameter. When this constructor is used, operations that are ready to be processed are scheduled on the second scheduler rather than being processed immediately.

Declarations

```
#define kTInterruptSchedulerID   "slm:sked$insk,1.1"


                    TInterruptScheduler();
                    TInterruptScheduler(TScheduler*,
                              unsigned long priority);
virtual             ~ TInterruptScheduler();

virtual   Boolean   IsValid() const;

virtual   void      Schedule(TOperation*);
```

Member functions

IsValid

The IsValid member function returns true if the TInterruptScheduler object was initialized properly after it was created. Call IsValid after creating a TInterruptScheduler to verify that it was constructed correctly. If it returns false, the scheduler should be deleted and not used. This will be the case when creating a TInterruptScheduler on a MacPlus running System 6.

Continued on following page ▶

`Schedule`

The `Schedule` member function schedules a `TOperation` object. The newly scheduled `TOperation` object will not be processed until deferred task time.

See also

`TOperation`
`TScheduler`
`TInterrupt`
`TPriorityScheduler`

`SchedulerExample` on the *ASLM Examples* disk

TIterator

The TIterator class lets you iterate through all objects in a TCollection object.

The TIterator class has the following inheritance:

```
TDynamic  -->  TIterator
```

Description

You need to use TIterator when you do not know what kind of data structure is being used for a TCollection or do not have access to the actual data (which should always be the case unless you are implementing a TCollection subclass). You can call the TCollection::CreateIterator member function to create a TIterator object for a collection. All TCollection subclasses have a TIterator subclass that is capable of iterating over the collection.

By using an object of the TMatchObject class—which can tell you whether two objects are equal—you can determine which objects an iterator should return. For example, assume that you want an iterator to return only employee records for employees in a certain pay range. To carry out this operation, you can set the TMatchObject for the iterator to a TMatchObject whose Compare member function looks at the employee's salary and returns 0 (0 indicates a match) only for employees within the specified pay range. The TIterator Next member function passes each employee in the collection to the TMatchObject Compare member function and returns the first object for which the Compare member function returns 0. This lets TMatchObject act as a filter for TIterator. If you want your iterator to use a match object, you can call SetMatchObject to set TMatchObject for the iterator.

Declarations

```
#define kTIteratorID "!$iter,1.1"

virtual            ~ TIterator();

virtual   void      Reset()            = 0;
virtual   void*     Next()             = 0;

virtual   Boolean   IterationComplete() const     = 0;
virtual   Boolean   RemoveCurrentObject()         = 0;

          void      SetMatchObject(TMatchObject*);
                    TMatchObject*  GetMatchObject() const;
```

Continued on following page ▶

Member functions

GetMatchObject

The GetMatchObject member function returns the match object that was set with SetMatchObject.

IterationComplete

The IterationComplete member function returns true if the iterator has finished iterating, and returns false if an iteration has stopped because the iterator has become invalid. A TIterator becomes invalid if there is a change in the contents of the TCollection object while iteration is in progress. The contents of the TCollection object can be changed by an interrupt that takes place during iteration. The contents of a collection can also be changed by the code that is doing the iterating. However, a call to RemoveCurrentObject does not make the iterator invalid, even though it changes the contents of the collection. If the iterator does become invalid, you can call Reset and restart the iteration.

Next

The Next member function returns the next object in the iteration, or NULL if the iteration is complete or has become invalid.

Since collections are thread-safe, when the Next member function returns NULL, you should call IterationComplete. If it returns true, then you were returned NULL because the iterator was done. Otherwise, you were returned NULL because the underlying collection changed.

RemoveCurrentObject

The RemoveCurrentObject member function removes the current object in the iteration (the object that Next most recently returned). RemoveCurrentObject returns false if the removal failed (most likely because the collection has changed and the iterator has become invalid.)

Reset

The Reset member function restarts an iteration from the beginning. You can call Reset when a TIterator object becomes invalid or when you just want to iterate through a TCollection object again.

SetMatchObject

The SetMatchObject member function sets the TMatchObject for TIterator.

See also TMatchObject

TLibraryFile

This is the C++ front-end for the C routines that let a shared library access the resources in the shared library's file.

The `TLibraryFile` class has the following inheritance:

```
TDynamic  -->  TLibraryFile
```

Description

The `TLibraryFile` class allows you to access the resources in the shared library's file. It contains member functions to place the library file's resource fork in the resource chain so Resource Manager calls can work. The `TLibraryFile` class also provides member functions that serve as a front end to some operating system Resource Manager calls. These member functions keep track of the use of resources so clients and libraries can share the resources.

IMPORTANT The `TLibraryFile` resource management calls are not interrupt-safe and are not meant to be portable; in fact, they may not exist on non-Macintosh systems.

Declarations

```
#define kTLibraryFileID "!$lfil,1.1"


virtual   Ptr  GetSharedResource(ResType, int the ID,
                                        OSErr* = NULL) = 0;
virtual   Ptr  GetSharedIndResource(ResType, int index,
                                        OSErr* = NULL) = 0;
virtual   Ptr  GetSharedNamedResource(ResType,
                                              const char* name,
                                              OSErr* = NULL) = 0;


virtual   void    ReleaseSharedResource(Ptr) = 0;
virtual   long    CountSharedResources(ResType) = 0;

virtual   size_t  GetSharedResourceUseCount(Ptr) const = 0;
virtual   OSErr   GetSharedResourceInfo(Ptr,
                      size_t* theSize = NULL,
                      short* theID = NULL, ResType* = NULL,
                      char* theName = NULL) const = 0;

virtual   long          GetRefNum() const = 0;
virtual   TFileSpec*    GetFileSpec() const = 0;
```

Continued on following page ▶

```
virtual   OSErr          OpenLibraryFile() = 0;
virtual   OSErr          CloseLibraryFile() = 0;

virtual   OSErr          Preflight(long& savedRefNum) = 0;
virtual   OSErr          Postflight(long savedRefNum) = 0;
```

Member functions

The C versions of the member functions are described in “Library File and Resource Management” in Chapter 7, “ASLM Utilities.” They have the same names and parameters as the `TLibraryFile` member functions, except that they require you to pass the `TLibraryFile` to act on as a parameter.

TLibraryID

The `TLibraryID` class is used to identify shared libraries.

The `TLibraryID` class has no parent class.

Description

A `TLibraryID` object, like a `TClassID` object, is a C string made up of a developer ID and a library name separated by a dollar sign (`$`), and optionally followed by version information.

Library IDs are written in this format:

```
xxxx$MyLibrary[,1.2.3]
```

Although the version is not required, your library ID should contain a version number so each release of your library will have unique library ID. It also allows you to reuse the same library ID (with a new version number) with a future version of the library. See Appendix D, "Versioning," for more information on version numbers in library IDs.

Declarations

`TLibraryID` is typedef'd to be the same as `TClassID.`

```
#define        TLibraryID          TClassID
```

See also

`TClassID`

Appendix D, "Versioning"

TLibraryManager

The `TLibraryManager` class is the interface that clients and shared libraries use to access many ASLM functions.

The `TLibraryManager` class has the following inheritance:

```
TDynamic  -->  TLibaryManager
```

Description "Creating and Deleting the Local Library Manager" in Chapter 7, "ASLM Utilities," provides more information on how to create and delete a `TLibraryManager` object and how it is used.

All `TLibraryManager` member functions have C-language equivalents. These C functions have the same names and parameters as the `TLibraryManager` member functions, with the exception of the `NewObject` and `GetFunctionPointer` functions. Since there is more than one version of each of these functions, different names are needed for the C-language equivalents.

Most of the member functions of this class are described in Chapter 7, "ASLM Utilities," using the C-language equivalents. All the C versions of `TLibraryManager` member functions use the local library manager.

Declarations

```
#ifdef __cplusplus

#define kTLibraryManagerID "!$lmgr,1.1"

virtual     void*       NewObject(const TClassID& classID,
                              OSErr* = NULL, TStandardPool* = NULL)
                              const;
virtual     void*       NewObject(const TClassID& classID,
                              const TClassID& baseClassID,
                              OSErr* = NULL, TStandardPool* = NULL)
                              const;
virtual     void*       NewObject(const TFormattedStream&,
                              OSErr* = NULL, TStandardPool* = NULL)
                              const;

virtual     TClassInfo* GetClassInfo(const TClassID&, OSErr* = NULL)
                              const;
```

```
virtual     OSErr       VerifyClass(const TClassID& classID,
                              const TClassID& baseClassID) const;
virtual     void*       CastObject(const void* obj,
                              const TClassID& parentID,
                              OSErr* = NULL) const;
virtual     void*       CastToMainObject(const void* obj) const;

virtual     OSErr       LoadClass(const TClassID&,
                              BooleanParm loadAll = false);
virtual     OSErr       UnloadClass(const TClassID&);
virtual     Boolean     IsClassLoaded(const TClassID&) const;

            OSErr       LoadFunctionSet(const TFunctionSetID&,
                              BooleanParm loadAll = false);
            OSErr       UnloadFunctionSet(const TFunctionSetID&);
            Boolean     IsFunctionSetLoaded(const TFunctionSetID&)
                              const;

virtual     ProcPtr     GetFunctionPointer(const TFunctionSetID&,
                              const char* funcName, OSErr* = NULL);
virtual     ProcPtr     GetFunctionPointer(const TFunctionSetID&,
                              unsigned int index, OSErr* = NULL);

virtual     OSErr       LoadLibraries(BooleanParm forceAll = true,
                              BooleanParm doSelf = true);
virtual     OSErr       UnloadLibraries();
virtual     void        ResetFunctionSet
                              (const TFunctionSetID* = NULL);

virtual     Boolean     TraceLogOn();
virtual     Boolean     TraceLogOff();

virtual     void        RegisterDynamicObject(TDynamic*);
virtual     void        UnregisterDynamicObject(TDynamic*);

            void              SetObjectPool(TStandardPool*);
            TStandardPool*    GetObjectPool() const;
            void              SetDefaultPool(TStandardPool*);
            TStandardPool*    GetDefaultPool() const;
            GlobalWorld       GetGlobalWorld() const;
virtual     TLibrary*         GetLibrary() const;
virtual     TLibraryFile*     GetLibraryFile() const;
```

Continued on following page ▶

Member functions

`CastObject`
`CastToMainObject`

The `CastObject` and `CastToMainObject` member functions are described in “Verifying an Object’s Type” in Chapter 7, “ASLM Utilities.”

`Dump`

The `Dump` member function displays a list of all known classes in the TraceMonitor Trace window.

`GetClassInfo`

The `GetClassInfo` member function returns information about a base class and the classes that inherit from it. It returns a `TClassInfo` object, which is a `TIterator` subclass and is used to iterate through the desired subclasses. See “TClassInfo” for more information on `GetClassInfo`.

`GetDefaultPool`
`SetDefaultPool`

For information about `GetDefaultPool` and `SetDefaultPool`, see “Memory Management Classes” in Chapter 8, “ASLM Utility Class Categories.”

`GetFunctionPointer`

The `GetFunctionPointer` member function obtains a pointer to a function. For more information about `GetFunctionPointer`, see “Calling Functions by Name” in Chapter 7, “ASLM Utilities.” Since `GetFunctionPointer` is overloaded, there are two C-language equivalents: `GetFunctionPointer` and `GetIndexedFunctionPointer.`

`GetGlobalWorld`

The `GetGlobalWorld` member function returns the global world for the client owning the `TLibraryManager` object. For additional information, see “Global World Functions” in Chapter 7, “ASLM Utilities.”

`GetLibrary`

The `GetLibrary` member function returns the `TLibrary` object for the shared library owning the `TLibraryManager` object. It returns `NULL` when it is called for a `TLibraryManager` object that was created for a non-shared library client, such as an application.

GetLibraryFile

The `GetLibraryFile` member function returns the `TLibraryFile` object for the shared Library owning the `TLibraryManager` object. It returns `NULL` when it is called for a `TLibraryManager` object that was created for a non-shared library client, such as an application.

IsClassLoaded
IsFunctionSetLoaded
LoadClass
UnloadClass
LoadFunctionSet
UnloadFunctionSet
LoadLibraries
UnloadLibraries

These member functions are all described in “Loading and Unloading Shared Libraries” in Chapter 7, “ASLM Utilities.”

NewObject

The `NewObject` member function is described in “Using NewObject” in Chapter 7, “ASLM Utilities.” Since `NewObject` is overloaded, there are three C-language equivalents: `NewObject`, `NewObjectWithParent`, and `NewObjectWithStream`. Like its `TLibraryManager` equivalent, `NewObjectWithStream` is not yet supported in version 1.1 of the ASLM.

RegisterDynamicObject
UnregisterDynamicObject

The `RegisterDynamicObject` and `UnregisterDynamicObject` member functions register any object that is to appear in the Inspector application. These member functions are described in “Registering C++ Objects With the Inspector” in Chapter 7, “ASLM Utilities.”

ResetFunctionSet

The `ResetFunctionSet` member function is described in the “Loading and Unloading Shared Libraries” in Chapter 7, “ASLM Utilities.”

SetObjectPool
GetObjectPool

The `SetObjectPool` and `GetObjectPool` member functions set and get the object pool associated with the `TLibraryManager` object. This pool is the same as the local pool. For more information, see “Memory Management Classes” in Chapter 8, “ASLM Utility Class Categories.”

Continued on following page ▶

`TraceLogOn`
`TraceLogOff`

The `TraceLogOn` and `TraceLogOff` member functions control the global `TTraceLog` object’s tracing operations. When tracing is on, the output of `Trace` is displayed in the TraceMonitor Trace window.

`VerifyClass`

The `VerifyClass` member function is described in “Verifying a Class’s Base Class” in Chapter 7, “ASLM Utilities.”

TLink

The `TLink` class implements a link object that can be placed on a linked list.

The `TLink` class has no parent class.

Description

The `TLink` class is used primarily to maintain `TLinkedList` objects. It holds a pointer to an object and a pointer to the next link in the list.

Because `TLink` is completely non-virtual and is only eight bytes long, it is very fast and efficient. A `TLink` object is often a field of the object to which it points.

Declarations

```
                  TLink(void* value);


                  TLink(TLink* link, void* value);
                  TLink(BooleanParm);
                  TLink();
                  ~TLink();
void* operator new(size_t size, TMemoryPool*);  // default size from a
                                                // pool
void* operator new(size_t);                     // from default pool
void  operator delete(void* mem);


void        SetNext(TLink* link);
TLink*      GetNext() const;

void*       GetValue() const;
void        SetValue(void*);

void        Append(TLink* newLink);  // append newLink after this
void        Remove(TLink* previous); // remove nextLink from list
```

Member functions

Append
Remove

The `Append` member function inserts the link passed to it into the list, and the `Remove` member function removes the link from the list. When you call `Remove`, you must specify in a parameter the link that precedes the link to be removed.

Continued on following page ▶

`SetNext`
`GetNext`
`SetValue`
`GetValue`

A `TLink` object has two fields: an `fNext` field that points to the next `TLink` object in a list, and an `fValue` field that points to the object. `SetNext` and `GetNext` set and get the `fNext` field, and `GetValue` and `SetValue` set and get the `fValue` field.

Instead of using `SetValue` to set the link's `fValue`, you can pass the object to the `TLink` constructor. If you do not want the constructor to initialize the link, you can pass `false` to the constructor. This makes constructing the link slightly faster.

See also `TLinkedList`

TLinkedList

The `TLinkedList` class is a `TSimpleList` subclass that adds the ability to do things with a linked list based on “after” or “before” rules.

The `TLinkedList` class has the following inheritance:

```
TDynamic --> TCollection --> TSimpleList --> TLinkedList
```

Description

The `TLinkedList` class provides several member functions in addition to those belonging to the `TSimpleList` class. Their names are intuitive and largely self-explanatory. `TLinkedList` does not have its own `TIterator` class because the `TListIterator` class also works for `TLinkedLists`. The `TListIterator` class iterates through the linked list.

Declarations

```
#define kTlinkedListID “slm:coll$11st, 1.1”


                              TLinkedList();
                              TLinkedList(TMemoryPool*);
                              TLinkedList(TSimpleList*);
virtual                       ~ TLinkedList();

// New member functions

virtual void*   After(const void* obj) const;
virtual void*   After(const TMatchObject&) const;
virtual void*   Before(const void* obj) const;
virtual void*   Before(const TMatchObject&) const;

virtual Boolean AddLinkAfter(TLink*, const TMatchObject&);
virtual Boolean AddLinkAfter(TLink*, const void* obj);
virtual Boolean AddLinkBefore(TLink*, const TMatchObject&);
virtual Boolean AddLinkBefore(TLink*, const void* obj);
virtual OSErr   AddAfter(void*, const TMatchObject&);
virtual OSErr   AddAfter(void*, const void* obj);
virtual OSErr   AddBefore(void*, const TMatchObject&);
virtual OSErr   AddBefore(void*, const void* obj);
```

Member functions

AddAfter
AddBefore

The `AddAfter` and `AddBefore` member functions return error codes other than `kNoError` if they fail to add the object to the list.

Continued on following page ▶

`AddLinkAfter`
`AddLinkBefore`

The `AddLinkAfter` and `AddLinkBefore` member functions return `false` if the link to be added before or after cannot be found.

`After`
`Before`

The `After` and `Before` member functions return the object that is located immediately after or before the object passed to it in the list.

See also

`TListIterator`
`TMatchObject`
`TSimpleList`
`TCollection`

TLinkedListExample on the *ASLM Examples* disk

TListIterator

The `TListIterator` class is used to iterate all collection classes descending from `TSimpleList`, including `TLinkedList` and `TPriorityList`.

The `TListIterator` class has the following inheritance:

```
TDynamic  -->  TIterator  -->  TListIterator
```

Description

For information on `TListIterator`, see “TIterator” earlier in this chapter.

Declaration

```
#define kTListIteratorID “!$litr, 1.1”

                         TListIterator(TSimpleList*);
virtual                  ~ TListIterator();

// TIterator Overrides

virtual   void           Reset();
virtual   void*          Next();

virtual   Boolean        IterationComplete() const;
virtual   Boolean        RemoveCurrentObject();

// New member functions

virtual   TLink*         GetCurrentLink() const;
          void           SetList(TSimpleList*);
```

Member functions

GetCurrentLink

The `GetCurrentLink` member function returns the `TLink` object of the current object (the object just returned by `Next`).

IterationComplete
Next
RemoveCurrentObject
Reset

For information on these functions, see “TIterator” earlier in this chapter.

Continued on following page ▶

`SetList`

The `SetList` member function is used to change the list you want to iterate over. It automatically calls `Reset` after changing the list.

See also

`TIterator`
`TSimpleList`

TSimpleListExample on the *ASLM Examples* disk

TMacFileSpec

The TMacFileSpec class keeps track of files by using a filename, volume refNum, and directory ID.

The TMacFileSpec class has the following inheritance:

```
TFileSpec  -->  TMacFileSpec
```

Description

The TMacFileSpec class is the only TFileSpec subclass supported in ASLM 1.1.

You can directly access or change the three fields used to specify the location of the library file. You can also use InitMacFileSpec to change the fields after creating the TMacFileSpec object.

For more information see "TFileSpec" earlier in this chapter.

Declaration

```
void InitMacFileSpec(TMacFileSpec *spec, int vRefNum, long parID,
                                                       Str63 name);

TMacFileSpec(const TMacFileSpec&);
TMacFileSpec(int vRefNum, long parID, Str63 name);

void* operator new(size_t, size_t fileNameLen,
                  TMemoryPool *thePool = NULL)
void* operator new(size_t size)
void  operator delete(void* obj)

short fVRefNum;   // volume refNum of volume file is on
long  fParID;     // dirID of the folder file is in
Str63 fName;      // name of the file
```

Member functions

operator new

The TMacFileSpec class provides an operator new override that allows you to specify the length of the filename (the default size is 63). This is useful for reducing the amount of memory TMacFileSpec occupies.

See also

TFileSpec

TMacSemaphore

The `TMacSemaphore` class implements a simple semaphore that can prevent data from being changed by another process while a client is trying to access it.

The `TMacSemaphore` class has the following inheritance:

```
TDynamic  -->  TMacSemaphore
```

Description

A semaphore is a flag that can protect a critical piece of data from being unexpectedly accessed by more than one process at the same time. The `TMacSemaphore` class provides a semaphore that can prevent data from being changed by interrupts.

Since the Macintosh Operating System supports only one thread of execution, the only way that data can be changed while you are trying to access it is for the data to be changed by an interrupt. Therefore, on the Macintosh, semaphores work by simply locking out interrupts.

Although this solution is simple, it can be dangerous. It means that you should never hold (or “grab”) the ASLM semaphore for more than a very short period of time. If you do, interrupts might be locked out for a dangerously long time—causing problems such as loss of network data.

This code fragment is an example of how you can use a `TMacSemaphore` object in an ASLM client:

```
TMacSemaphore* semaphore = new TMacSemaphore;
semaphore->Grab();
if (count == 0)
     count++;
semaphore->Release();
```

In this example, it is assumed that the client wants to increment `count` only if `count` is equal to 0. If the semaphore were not used in the example, `count` could be changed by an interrupt after it has been determined that `count == 0` but before the code in the example increments `count` in the statement `count++`.

If an interrupt changed the value of `count` in this way, the code shown in the example would increment `count` again. Thus, `count` would end up being equal to 2, when you really want it to be equal to 1.

By using the semaphore as shown in the example, you can prevent interrupts from occurring and performing unwanted actions such as unexpectedly changing the value of `count` when you do not want the value changed.

Declarations

```
#define kTMacSemaphoreID "!$sema,1.1"

                              TMacSemaphore();
virtual                       ~ TMacSemaphore();

virtual void                  Grab();
virtual void                  Release();
virtual Boolean               GrabNoWait();
```

Member functions

Grab

The `Grab` member function grabs the semaphore which causes the interrupts to be blocked out.

GrabNoWait

The `GrabNoWait` member function grabs the semaphore if it is not already grabbed and returns `true` if the grab is successful. On the Macintosh Operating System, `Grab` never blocks, so `GrabNoWait` never fails. Since the Macintosh Operating System has only a single thread of execution, it is impossible to try to grab the same semaphore more than once from outside the same thread of code.

Release

The `Release` member function releases the semaphore and reenables interrupts by returning the interrupt level to the state it was in before the matching grab was called. You must call `Release` for every `Grab.`

See also

TMacSemaphoreExample on the *ASLM Examples* disk

TMatchObject

The `TMatchObject` class gives users of a collection a way to determine whether two objects are equal, rather than just having the collection compare object pointers.

The `TMatchObject` class has the following inheritance:

`TDynamic-->TMatchObject`

Description

This object is the base class for any object which “knows” how to hash a specific object, as well as how to compare a second object to the specific object.

Using `TMatchObject` to determine whether two objects are equal can prevent, for example, two objects with the same name from appearing in a collection. Objects of the `TMatchObject` class can also be used to filter out objects that you do not want returned to you by a `TIterator` object.

Objects of the `TMatchObject` class can be useful when you want to perform operations such as `AddUnique` or `Member` on a `TCollection` object, but want something other than the object’s pointer to determine whether two objects are equal. `TCollection` member functions such as `AddUnique` and `Member` have versions that use a `TMatchObject` parameter to help determine if the object is already in the collection.

The `TMatchObject` subclass should know about a specific type of object that will be added to the collection that the `TMatchObject` subclass will be used with. An example of how to do this is given with the `IsEqual` member function description below.

Declarations

```
#define kTMatchObjectID "!$mobj,1.1"

virtual                       TMatchObject();

// Default implementation is to return 0
virtual unsigned long         Hash() const;

// Default implementation is to compare
// address of "this" with address of the object
virtual short                 Compare(const void*) const;

// Default implementation is to call Compare
virtual Boolean               IsEqual(const void*) const;
```

Member functions

Compare

The Compare member function returns zero if the object passed to it matches comparison criteria that are specified for the TMatchObject. It returns -1 if the match object is considered to be “greater,” and 1 if the object passed to Compare is considered to be “greater.” Subclasses of TMatchObject must override the Compare member function so it can properly compare the object passed to it with the information stored in the match object (such as a name).

IMPORTANT The implementation of Compare and IsEqual should be designed to execute as fast as possible, since a semaphore is held when Compare and Equal are called—and, on the Macintosh, this disables interrupts.

Hash

The Hash member function is used to speed up hash list operations that use match objects. For example, when used by the THashListIterator, it tells the iterator which hash buckets to examine. It will be used in a similar way by the THashList:: Member member function to speed up searches. See “THashListIterator” and “THashList” for more information regarding hash functions.

IsEqual

The TCollection member functions such as AddUnique pass each object in the collection to the match object’s IsEqual member function (one at a time) to see if the object is already in the collection. Also, TIterator subclasses pass each object in the collection to the match object’s IsEqual member function (one at a time) for filtering purposes.

The IsEqual member function returns true if the match object and the object passed are considered to be equal, and returns false otherwise. The default implementation of IsEqual is to simply call Compare and return true if Compare returns 0 and false otherwise. For this reason, you normally do not need to override IsEqual. However, since interrupts are normally disabled when IsEqual is called, you should override IsEqual if comparing objects for equality is a lot faster than calling Compare.

Continued on following page ▶

Suppose, for example, that you had a collection of `TName` objects and wanted to make sure that all the names in the collection were always unique. If you called `AddUnique(void*)` to add `TName` objects to the collection, the same name might appear in the collection more than once because more than one `TName` object might have the same name. That is because `AddUnique(void*)` uses object pointers to determine whether two objects are the same, and a collection does not know anything about names. You can do the following to avoid this problem:

1 Subclass `TMatchObject`. (For example, you can create a subclass named `TNameMatchObject`.)

2 Add a name field to your subclass and set the name field to the name of the `TName` object that you want to add to the collection.

3 Have the `TMatchObject` subclass's `Compare` member function compare the name field to the name in the `TName` object passed to it, returning 0 if they are equal, `-1` if the match object's name is greater, and `1` if the `TName` object's name is greater. (`Compare` must cast the `void*` passed to it to a `TName*`.)

4 Call the member function `TCollection::AddUnique(void*, TMatchObject&)` to add the object to the collection. (Note that `AddUnique(void*)` uses object pointers to determine if two objects are the same.)

As `TCollection::AddUnique` iterates through `TCollection`, it passes each object to your `TNameMatchObject::IsEqual` member function. The `IsEqual` function compares its `TNameMatchObject`'s name with the name of the `TName` object passed to it (usually by calling the `Compare` member function), returning `true` if the objects are equal and returning `false` if they are not.

If `TNameMatchObject::IsEqual` returns `false` for all `TName` objects passed to it, `AddUnique` adds the object to the `TCollection`.

See also TArrayExample and TLinkedListExample on the *ASLM Examples* disk

TMemoryPool

The TMemoryPool class is the abstract class from which memory allocators should descend.

The TMemoryPool class has the following inheritance:

```
TDynamic  -->  TMemoryPool
```

Descriptions

The TMemoryPool class is an abstract class used for all pools. Some TMemoryPool member functions are pure virtual member functions that must be overridden. Memory pools are used for high seed interrupt safe memory allocation. For more information on memory pools see “Memory Management Classes” in Chapter 8, “ASLM Utility Class Categories.”

On the Macintosh, TMemoryPool subclasses always allocate their pools by using NewPtr and then blocks of memory are allocated out of these pools whenever the user calls one of the member functions that allocates memory.

Creating and deleting pools

When you create a memory pool using the TMemoryPool new operator, you pass the amount of memory that you want to be made available from the pool in the poolSize parameter (the second size_t parameter of new). The size of the pool object is automatically passed in the first size_t parameter of new. The pool object and the pool memory that are available for allocation coexist in the same physical block of memory.

Remember that each chunk you allocate from the pool requires some overhead. The TMemoryPool subclasses define a constant for the chunk overhead size. You should estimate how many chunks you will want to allocate from the pool, multiply this by the constant, and add the result to the poolSize parameter when the pool is created. If you do not add enough overhead to cover the number of chunks that you intend to allocate from the pool, you may not be able to allocate all of them.

Continued on following page ▶

WARNING You cannot create pools at interrupt time, and you cannot add memory to pools by calling `AddMemoryToPool` at interrupt time. This is because `AddMemoryToPool` makes calls to the Macintosh Memory Manager that are not interrupt-safe. If the ASLM knows it is being used at interrupt time (usually when a call to `EnterInterrupt` is made), the `AddMemoryToPool` function does not attempt to "grow" the pool.

`TMemoryPool` subclasses *must* be created using the `new` operator. A pool should never be created as a stack object or as a data member of a class.

Another parameter that is required when you create a pool is the zone from which memory is allocated for the pool. The first version of `new` accepts a zone type. Possible zone types are `kSystemZone`, `kKernelZone`, `kApplicZone`, `kCurrentZone`, and `kTempZone`. The `kSystemZone` and `kKernelZone` types are the same; they cause memory allocations from the pool to be made from the system heap. The `kTempZone` uses temporary memory, and `kApplicZone` uses the application zone. The `kCurrentZone` type uses memory from the current zone. On the Macintosh Operating System, this is normally the application zone of the currently executing application. You can get and set the current zone by using the Macintosh Memory Manager calls `GetZone` and `SetZone`.

The second version of `new` accepts a pointer to a heap zone (a `THz*` on the Macintosh). If a `NULL` pointer is passed for this parameter, the ASLM uses temporary memory.

An optional parameter that you can specify when you create a pool is `MemoryType`. The value of `MemoryType` is either `kNormalMemory`, `kHoldMemory`, `kLockMemory`, or `kLockMemoryContiguous`.

The names of all these constants are based on virtual memory terms. *Held* memory is memory that is never paged out to disk. *Locked* memory is memory that is held and never moved in physical memory. *Locked contiguous* memory is memory that is locked and is also stored contiguously in physical memory. If the `MemoryType` parameter is not specified, the ASLM uses `kNormalMemory` by default.

When you delete a `TMemoryPool` object, the ASLM frees all the memory allocated for the pool, including any additional blocks of memory that may have been added by `AddMemoryToPool` calls.

Using pool notifiers

The TMemoryPool class provides a facility for notifying clients when the amount of available memory falls below or exceeds a certain level. You can use this facility to expand (*grow*) your pool by calling AddMemoryToPool or to shrink your pool by calling DownSizePool.

ASLM provides the TPoolNotifier class to assist in growing pools when they are low on memory. Subclasses of TPoolNotifier can be created to either change the behavior of the notifier when the pool is low on memory and to do something when the pool has too much free memory. For more information on pool notifiers and how they are used, see the description of the Allocate member function below and the TPoolNotifier class later in this chapter.

WARNING The TMemoryPool objects often fail to allocate or grow on memory for machines with virtual memory turned on when you specify kLockMemoryContiguous MemoryType, especially if the machine has little real memory. It may not be possible to make the range of memory physically contiguous if any of the pages in the range are already locked, or if there is not a contiguous block in real memory that is large enough. Therefore, if you must have a pool with locked contiguous memory, allocate it as early as possible, preferably at system startup, to increase the likelihood of finding enough contiguous memory. The pool may not be able to grow at a later time.

Declarations

```
#define kTMemoryPoolID "!$pool,1.1"

virtual                            ~ TMemoryPool();

      void* operator new(size_t size, size_t poolSize,
                  ZoneType zType, MemoryType mType = kNormalMemory)
      void* operator new(size_t size, size_t poolSize, void* zone,
                  MemoryType mType = kNormalMemory)
      void* operator new(size_t size)
      void  operator delete(void* ptr)

virtual     void*       Allocate(size_t)                  = 0;
virtual     void*       Reallocate(void*, size_t)         = 0;
virtual     void        Free(void*)                       = 0;
virtual     size_t      GetSize(void*) const              = 0;
```

Continued on following page ▶

```
virtual     Boolean     CheckPool() const                  = 0;
virtual     void        GetPoolInfo(PoolInfo&) const;
virtual     void        TracePoolInfo() const;

virtual     Boolean     AddMemoryToPool(size_t);
virtual     void        DownSizePool();
virtual     size_t      GetLargestBlockSize() const        = 0;
            size_t      GetCurrentPoolSize() const;

            void              SetNotifier(TPoolNotifier*);
            TPoolNotifier*    GetNotifier() const;
            void              SetNotifyMarks(size_t low,
                                          size_t high = (size_t)-1L);

static      TMemoryPool*      RecoverPool(void*);
static      void*             AllocateMemory(size_t);
static      void*             AllocateMemory(TMemoryPool*, size_t);
static      void*             ReallocateMemory(void*, size_t);
static      void              FreeMemory(void*);
static      size_t            GetMemorySize(void*);
```

Member functions

AddMemoryToPool

The `AddMemoryToPool` member function adds a specified amount of memory to the memory available to the pool. It allocates the memory that is added to the pool from the heap that was specified when the pool was created. The `AddMemoryToPool` object also uses the `MemoryType` specified when the pool was created. On the Macintosh Operating System, the memory to be added to the pool is allocated by calling `NewPtr`.

Allocate

The `Allocate` member function allocates a block of memory from the pool. When you call `Allocate`, you pass the size of the block you want as a parameter. If `Allocate` cannot find enough memory, it calls the pool's `TNotifier` object, and the notifier then has the option of freeing some memory. `Allocate` will continue to call the `TPoolNotifier` object's `Notify` member function as long as it continues to free up memory and there is still not enough memory for the allocation. If the pool notifier does not free up enough memory and `Allocate` was not called at interrupt time, then it will immediately grow the pool so it has enough memory, otherwise it will return `NULL`. See "TPoolNotifier" later in this chapter for more information on pool notifiers.

AllocateMemory
ReallocateMemory
FreeMemory
GetMemorySize

The `AllocateMemory` member function works like `Allocate`, but is a static function that takes the pool from which to allocate as a parameter. There is also a version of `AllocateMemory` that uses the pool returned by `GetDefaultPool` instead of taking the pool as a parameter. Other `TMemoryPool` static functions include `ReallocateMemory` (similar to `Reallocate`), `FreeMemory` (similar to `Free`), and `GetMemorySize` (similar to `Size`).

DownSizePool

The `DownSizePool` member function frees all memory that was added to the pool with `AddMemoryToPool` and does not currently have any blocks of memory allocated from it.

Free

The `Free` member function returns to the pool the block that is passed to it.

GetCurrentPoolSize

The `GetCurrentPoolSize` member function returns the current size of the pool.

GetSize

The `GetSize` member function returns the size of the block passed to it.

GetLargestBlockSize

The `GetLargestBlockSize` member function returns the largest block size that is available for allocation.

CheckPool

The `CheckPool` member function returns `true` if no problems are found with the pool. When you are debugging code, it is good practice to call `CheckPool` now and then to make sure that you are not corrupting the pool.

Continued on following page ▶

`GetPoolInfo`

The `GetPoolInfo` member function returns a `PoolInfo` data structure that contains the number of free bytes in the pool (`fFreeBytes`), the size of the largest block in the pool (`fLargestBlock`), the "high-water mark" that shows the most memory that has been used at the same time from the pool (`fMaxUsage`), and the current size of the pool, including both free and allocated blocks (`fCurSize`).

Note: To see if you have enough memory to allocate a block, you must check `fLargestBlock`—not `fFreeBytes`—because the pool's memory may be fragmented. The value stored in `fMaxUsage` is the maximum amount of memory that has been allocated from the pool at any one time, including per-block overhead and the extra memory that must be allocated during a `Reallocate` call. You can use the value of `fMaxUsage` as a guideline to help you figure out how big your pool should be. In deciding how big to make your pool, you also should consider how fragmented your pool may become. The pool may become fragmented during the normal course of allocating and reallocating blocks because pool memory is non-relocatable and cannot be compacted.

`Reallocate`

The `Reallocate` member function reallocates a block of memory to a new size that can be larger or smaller than the original size. When you call `Reallocate`, you pass the member function two parameters: a pointer to the block that you want to reallocate and the new size that you want allocated. A `Reallocate` call can fail if there is not enough memory in the pool for both the original block and the new block.

When you call `Reallocate` to reallocate memory to a larger block, the `Reallocate` function attempts to merge the block with any free block before or after it. If this is not possible, `Reallocate` must be able to store both the original block and the new block in memory at the same time.

`RecoverPool`

The `RecoverPool` member function returns the `TMemoryPool` object that was used to allocate a specified block of memory.

SetNotifyMarks
SetNotifier
GetNotifier

The `SetNotifyMarks` member function sets the low and high free memory marks that the pool's notifier will be warned about. The `SetNotifier` function specifies the `TPoolNotifier` object to use for notification when the low or high mark is reached. The `GetNotifier` function returns the `TPoolNotifier` for the pool. When your notifier is called, you can schedule a `TOperation` on a `TTaskScheduler` to allocate more memory. For more information on pool notifiers, see "Using Pool Notifiers" earlier in this section.

TracePoolInfo

The `TracePoolInfo` member function writes the information obtained by `GetPoolInfo` to the TraceMonitor's Trace window.

See also

`TPoolNotifier`
`TStandardPool`
`TChunkyPool`

TPoolNotifierExample on the *ASLM Examples* disk

TMethodNotifier

The TMethodNotifier class is the base class for notifiers that call a member function in an object.

The TMethodNotifier class has the following inheritance:

```
TDynamic -->  TNotifier  -->  TMethodNotifier
```

Descriptions

A TMethodNotifier object uses a member function of an object as the callback, so it is more object–oriented than TProcNotifier. The constructor takes a pointer to the member function to call for notification and the object that the member function belongs to. Although the object passed to the constructor is declared as a TDynamic*, the only requirement is that it inherit from SingleObject and that it have its v-table first.

Declarations

```
#define kTMethodNotifierID "!$mnot,1.1"


               TMethodNotifier(TDynamic*, NotifyMethod);
               TMethodNotifier(const TMethodNotifier&);
virtual        ~ TMethodNotifier();

virtual void   Notify(EventCode, OSErrParm = kNoError,
                    void* notifyData = NULL);

TDynamic*      GetObject() const;
```

Member functions

GetObject

The GetObject member function returns the object associated with the TMethodNotifier object. A TMethodNotifier object has an object pointer that is attached by the creator of the notifier and is returned as a TDynamic* by calling GetObject. The TMethodNotifier's NotifyMethod must point to a member function of the object associated with the notifier.

Notify

The Notify member function calls the NotifyMethod that was passed to the constructor when the object was created. The Notify function sets the global world to the global world stored with the TMethodNotifier object when it was created and sets the current client to the client that owns the global world. It then calls the NotifyMethod that was set up when the TMethodNotifier object was created.

See also	`TNotifier` `TProcNotifier`
	TMethodNotifierExample on the *ASLM Examples* disk

TMicroseconds

This `TTime` subclass is used to specify an initial time value in microseconds—that is, it provides a constructor that takes a time value in microseconds.

The `TMicroseconds` class has the following inheritance:

```
TDynamic  -->  TMatchObject  -->  TDoubleLong  -->
                              TTime  -->  TMicroseconds
```

Description

For additional information, see “TTime” later in this chapter.

Declarations

```
#define kTMicrosecondsID "slm:supp$mics,1.1"

                    TMicroseconds();
                    TMicroseconds(unsigned long msecs);
                    ~ TMicroseconds();

                    operator unsigned long() const;
virtual   double    ConvertToDouble() const;
                    operator double() const;
```

Member functions

operator unsigned long

The `operator unsigned long` member function returns the number of microseconds in an `unsigned long`.

ConvertToDouble

The `ConvertToDouble` member function converts the time to a `double` containing the number of microseconds.

operator double

The `operator double` member function returns the number of microseconds in a `double` by calling `ConvertToDouble`.

See also

`TTime`

TTimeExample on the *ASLM Examples* disk

TMilliseconds

This TTime subclass is used to specify an initial time value in milliseconds—that is, it provides a constructor that takes a time value in milliseconds.

The TMilliseconds class has the following inheritance:

```
TDynamic  -->  TMatchObject  -->  TDoubleLong  -->
                              TTime  -->  TMilliseconds
```

Description

For additional information, see “TTime” later in this chapter.

Declarations

```
#define kTMillisecondsID "slm:supp$mils,1.1"

                    TMilliseconds();
                    TMilliseconds(unsigned long msecs);
                    ~ TMilliseconds();

                    operator unsigned long() const;
virtual double      ConvertToDouble() const;
                    operator double() const;
```

Member functions

operator unsigned long

The `operator unsigned long` member function returns the number of milliseconds in an `unsigned long`.

ConvertToDouble

The `ConvertToDouble` member function converts the time to a `double` containing the number of milliseconds.

operator double

The `operator double` member function returns the number of milliseconds in a `double` by calling `ConvertToDouble`.

See also

`TTime`

TTimeExample on the *ASLM Examples* disk

TNotifier

The TNotifier class and its subclasses are used for asynchronous notification of events.

The TNotifier class has the following inheritance:

```
TDynamic  -->  TNotifier
```

Description

The TNotifier class is used as a base class for classes that are used for asynchronous notification of events. The ASLM provides two general purpose TNotifier subclasses: TProcNotifier and TMethodNotifier. The TProcNotifier handles notification by calling a C function that is passed to the TProcNotifier object when it was constructed. The TMethodNotifier handles notification by calling a method of an object, both of which are passed to the TMethodNotifier object when it is constructed. The ASLM also provides the TPoolNotifier class which is used by TMemoryPool subclasses for notification when the memory pool has either too little or too much free memory.

The TNotifier constructor saves the current global world in a field that can then be accessed by the TNotifier subclass. Its main use is by the Notify member function for setting up the global world and the current client.

Declarations

```
typedef unsigned long         EventCode;

#define kTNotifierID "!$noti,1.1"

               TNotifier();
virtual        ~ TNotifier();

virtual void   Notify(EventCode, OSErrParm = kNoError,
                    void* = NULL) = 0;
```

Member functions

Notify

The Notify member function is called to notify the TNotifier subclass of an asynchronous event. The contents of the three parameters passed to Notify are up to the caller, but usually there will be some agreement between the caller and the TNotifier subclass on what the parameters will contain.

The `TNotifier` has a `GlobalWorld` field that is set up to be equal to the current global world when an instance of the `TNotifier` subclass is created. This is handled automatically by the `TNotifier` constructor. The subclass's `Notify` member function can use this field to set up the global world and current client when it is called.

See also

`TMethodNotifier`
`TProcNotifier`

TProcNotifierExample and TMethodNotifierExample on the *ASLM Examples* disk

TOperation

A `TOperation` object contains the implementation of a task to be performed.

The `TOperation` class has the following inheritance:

```
TDynamic  -->  TOperation
```

Description

The `TOperation` objects (containing the implementation of a task to be performed) are normally placed on `TScheduler` objects so that they can be executed at a later time. But they also have other uses, such as being used in place of callback procedures.

When an operation is ready to be processed, its `Process` member function is called. It is up the scheduler that the operation is on to decide when it should be processed. The default implementation of the `Process` member function is to call the operation's `ProcessProc` if it has one. The `ProcessProc` is simply a C function that is set up when the operation is constructed. Subclasses of `TOperation` may choose to make the `Process` member function do all the work rather than calling the `ProcessProc`. See `Process` below for more information.

`TOperation` objects can have reference data stored with them so the operation has some context when it is called. Often this reference data is simply a pointer to the object that created the `TOperation` .

Setting up a global world for an operation

`TOperation` objects have the ability to have their global world (or any other global world) be set up as the current global world when the operation is processed. There are two ways to set up the global world for an operation. One strategy is to store the global world with the operation. The other way is to store the global world with an ASLM scheduler such as `TTimeScheduler` or `TInterruptScheduler`. In either case, the scheduler sets up the global world before processing the operation, and then restores it afterwards. It will also set the current client to the client that owns the global world by using the `SetClientToWorld` routine. (For more information about the ASLM scheduler classes, see “Process Management Classes” in Chapter 8, “ASLM Utility Class Categories.”)

If the global world of the scheduler that you use is set to the constant `kInvalidWorld` (which is the default), the scheduler sets the current global world to the operation's global world before processing the operation, unless the operation's global world is also set to `kInvalidWorld`. Then, the current global world is not changed. If the scheduler's global world is not set to `kInvalidWorld`, the scheduler sets the global world to the scheduler's global world before processing the operation.

All operations have their saved global world set to the current global world when they are created. You can set an operation's saved global world by calling `SetSavedGlobalWorld`. You can retrieve an operation's saved global world by calling `GetSavedGlobalWorld`.

Setting a scheduler's global world

All schedulers have their global world set to the constant `kInvalidWorld` when they are created. You change a scheduler's global world by calling `TScheduler::SetSchedulerWorld`. You can retrieve the scheduler's global world by calling `TScheduler::GetSchedulerWorld`. `TScheduler::IsSchedulerWorldValid` returns `false` if the scheduler's global world is set to `kInvalidWorld`. Otherwise, `TScheduler::IsSchedulerWorldValid` returns `true`.

Since the default is for schedulers to have their global world set to `kInvalidWorld` and for operations to have their global world set to the world that was current when they were created, the ASLM default behavior is for an operation to be processed in the world that was current when it was created.

Usually, this default behavior is satisfactory; that is, you can usually create your `TScheduler` and `TOperation` objects without doing anything special to get the global world set up properly when your operation is processed. However, you might want to change an operation's global world if the implementation of the `ProcessProc` is in a different global world than the code that created the operation.

For example, someone might give you an operation whose `ProcessProc` you get to set before scheduling it. Also, you might want to set the scheduler's global world if the scheduler is run by one client, the operations on the scheduler were created by a second client, and the operation's `ProcessProc` or `Process` member function calls code belonging to the first client. If the operation's `ProcessProc` or `Process` member function does not care about the global world, but the code that it calls in the first client does care, then the first client should set the scheduler's global world to its own world. This is a rare case, but it does sometimes turn up—for example, in the Inspector application.

Declarations

```
#define kTOperationID         "!$oper,1.1"

#define kRemovedInProcess ((TPriorityLink*)-1L)
```

Continued on following page ▶

```
               TOperation();
               TOperation(long creatorData);
               TOperation(void* creatorPtr);
               TOperation(ProcessProc, long creatorData);
               TOperation(ProcessProc, void* creatorPtr);
               TOperation(const TOperation&);
virtual        ~ TOperation();

     TPriorityLink* GetLink();

virtual void   Reset();
virtual void   Process();

     Boolean   WasRemovedInProcess() const;
     void      ClearRemovedInProcess();
     void      SetDeleteWhenDone();
     Boolean   IsBeingRerun() const;
     void           SetProcessProc(ProcessProc);
     ProcessProc    GetProcessProc() const;

// Timer and Priority are just two different ways
// of looking at the same field.

     void           SetTime(const TTime&);
     void           SetTime(unsigned long msecs);
     void           SetPriority(unsigned long pri);
     unsigned long  GetTime() const;
     unsigned long  GetPriority() const;

// CreatorData and CreatorPtr are just 2 different ways
// of looking at the same field.

     void*          GetCreatorPtr() const;
     long           GetCreatorData() const;
     void           SetCreatorPtr(void*);
     void           SetCreatorData(long);

     GlobalWorld    GetSavedGlobalWorld() const;
     void           SetSavedGlobalWorld(GlobalWorld);
```

Member functions

`GetLink`

All `TOperation` objects have a `TLink` field that is automatically initialized to point to the `TOperation` object. `GetLink` returns a pointer to this `TLink`. Normally it is only used by `TScheduler` subclasses for keeping the operation on a list of operations. Since the `TOperation` has only one `TLink` field, it can only be on one scheduler at a time.

`GetTime`
`SetTime`
`GetPriority`
`SetPriority`

The `TOperation` objects can be associated with a time if they are on a `TTimeScheduler` object or with a priority if they are on a `TPriorityScheduler` object (both of these classes are described later in this chapter). You can get and set the time by calling `SetTime` and `GetTime`. You can get and set the priority by calling `GetPriority` and `SetPriority`. A `TOperation` object cannot have both a time and a priority because the two values are stored in the same field.

`Process`

When you want to schedule a `TOperation`, you can pass the `TOperation` object to the `TScheduler::Schedule` member function. At some point, the `TScheduler::Run` member function is called, causing each `TOperation` to be removed from `TScheduler` and the `TOperation::Process` member function to be called for each operation.

IMPORTANT A `TOperation` cannot be scheduled a second time until it has been removed from the scheduler. This is because scheduled operations are maintained on a linked list and the link is part of the `TOperation` object. Therefore, a `TOperation` object can be on only one linked list at a time. A `TOperation` object can be rescheduled in its `ProcessProc` or `Process` member function because the object will already be removed when the `ProcessProc` or `Process` is called. Alternatively, a `TOperation` object can be rescheduled at any time after the operation has been processed.

There are two ways to control what a `TOperation` does when its `Process` member function is called.

- You can subclass `TOperation` and override its `Process` member function. This also gives you the opportunity to add more fields to the `TOperation`.
- You can set the `TOperation` object's `ProcessProc`.

Continued on following page ▶

The `ProcessProc` is a function that is called when the `TOperation` is processed. The default behavior of `TOperation::Process` is to call the `TOperation` object's `ProcessProc`. In fact, it is an error not to set the `ProcessProc` if you have not overridden the `Process` member function.

You can set the `ProcessProc` by either passing the appropriate parameter to the constructor or by calling `SetProcessProc`. The purpose of the `ProcessProc` is to let you use `TOperation` without having to subclass it and override the `Process` member function. In general, if you want to subclass `TOperation` to add more fields, you should override the `Process` member function, not set the `ProcessProc`.

Note: You can delete a `TOperation` object in its `Process` member function or its `ProcessProc`. Then the operation's creator does not need to keep track of the operation.

Reset

The `Reset` member function sets the operation's `ProcessProc` to `NULL` so that it will not be called when the operation is processed.

SetCreatorPtr
GetCreatorPtr
SetCreatorData
GetCreatorData

Each `TOperation` has a user data field, which you can use for any purpose you like. You can get and set the contents of the user data field by calling `SetCreatorPtr` and `GetCreatorPtr`, with a `Ptr` type passed as a parameter. Alternatively, you can get and set the user data field by calling `SetCreatorData` or `GetCreatorData`, with a `long` data type passed as a parameter. The `SetCreatorPtr` and `SetCreatorData` functions set the same field in a `TOperation` object.

One common use of the user data field is to set it to point to an object, possibly the one that created the `TOperation` object. You can also set the user data field by passing the appropriate parameter or parameters to the constructor.

SetProcessProc
GetProcessProc

The `SetProcessProc` member function is used to set the operation's `ProcessProc`, and `GetProcessProc` returns the operation's `ProcessProc`. You can also set the `ProcessProc` by passing the appropriate parameters to the constructor.

The purpose of the `ProcessProc` is to let you use `TOperation` objects without having to subclass them and override the `Process` member function. In general, if you want to subclass `TOperation` to add more fields, you should override the `Process` member function, not set the `ProcessProc`.

SetSavedGlobalWorld
GetSavedGlobalWorld

All `TOperation` subclasses have a global world associated with them that is set by the `TOperation` constructor to be equal to the current global world at the time of the operation's construction. `GetSavedGlobalWorld` is used to get this global world and `SetSavedGlobalWorld` is used to change its value. See "Setting up a Global World for an Operation" and "Setting up a Scheduler's Global World" earlier in this section to see how the global world is used.

WasRemovedInProcess
ClearRemovedInProcess
SetDeleteWhenDone
IsBeingRerun

These four member functions are used by clients of the `TTimeScheduler` class to coordinate removal of an operation-in-progress.

The `WasRemovedInProcess` function returns a flag that shows whether the current `TOperation` object was removed from its `TTimeScheduler` object while its `Process` routine was executing.

You can call `ClearRemovedInProcess` when a `TOperation` client determines that an operation that is in process has been removed from the `TTimeScheduler` object, and the `TOperation` object intends to delete itself.

You can call `SetDeleteWhenDone` when a `TOperation` object needs to be deleted. The `SetDeleteWhenDone` function should be called only when auto-rescheduling is `true` and the object being deleted has not been removed in process. (The object should first ensure that its time field is zero.)

You can call `IsBeingRerun` when it is important that the `TTimeScheduler` object does not look at a `TOperation` object's memory when the call returns. Typically, `IsBeingRerun` is called when `TOperation` is an embedded object and the parent object is to be deleted. In this case, `TOperation` should delete only the parent object when `IsBeingRerun` returns `true`.

Continued on following page ▶

The `WasRemovedInProcess`, `ClearRemovedInProcess`, `SetDeleteWhenDone`, and `IsBeingRerun` functions are described in more detail in “TTimeScheduler” later in this chapter.

See also `TScheduler`

TPoolNotifier

The `TPoolNotifier` class is used by `TMemoryPool` objects so that they can be notified when the pool reaches a low or high mark.

The `TPoolNotifier` class has the following inheritance:

```
TDynamic  -->  TNotifier  -->  TPoolNotifier
```

Description

The `TPoolNotifier` class can assist in automatically increasing the size of (growing) a pool when the pool comes dangerously close to running out of memory.

When you create a `TPoolNotifier` object, you pass to the constructor the percentage by which you want the pool to grow and the minimum number of bytes by which it should grow. You can then attach the notifier to one pool by calling the `TMemoryPool::SetNotifier` member function.

If you omit both constructor parameters, a default 10 percent "grow by" is used, with a 128-byte minimum grow.

See also `TGrowOperation` for information on "growing" a pool.

IMPORTANT A `TPoolNotifier` object can be attached to only one pool.

Declarations

```
#define kTPoolNotifierID      "!$plnt,1.1"


               TPoolNotifier(unsigned int growBy = 10,
                    unsigned int minGrow = 128);
virtual        ~ TPoolNotifier();

virtual void   Notify(EventCode, OSErrParm = kNoError,
                    void* = NULL);
virtual size_t GrowBy(TMemoryPool*, size_t);
```

Member functions

GrowBy

If a pool has a notifier attached to it and the pool does not have enough memory for `Allocate`, then `Allocate` attempts to grow the pool immediately if it is not interrupt time. The `Allocate` function calls `TMemoryPool::AddMemoryToPool` and uses the `TPoolNotifier::GrowBy` member function to determine the number of bytes to add to the pool.

Continued on following page ▶

The `GrowBy` member function returns the largest of these three sizes:

- the size passed to `GrowBy`
- the minimum amount the pool should grow by
- the percentage that the pool should grow by times the current size of the pool

The latter two sizes are determined by the parameters passed to the `TPoolNotifier` object's constructor.

The size passed to `GrowBy` is the size passed to the `Allocate` member function. This behavior ensures that the pool always grows by at least the amount by which the notifier would have grown the pool. It also makes sure that the pool grows by enough to handle the size being allocated.

Notify

You can call `TMemoryPool::SetNotifyMarks` to tell the pool when it should call the notifier's `Notify` member function to indicate that pool has reached a low or high mark.

The `TPoolNotifier` subclass only handles events in the category `kLowPoolMemoryEvent` (which occurs when the amount of memory in the pool goes below the low mark). The default behavior of the `Notify` member function for this event is to grow the pool immediately if it is not being called at interrupt time. If the member function is called at interrupt time, it schedules a `TGrowOperation` on the global `TTaskScheduler`. The `TGrowOperation` will then grow the pool at System Task time. If you want to do something special for `kHighPoolMemoryEvent`, you can subclass `TPoolNotifier` and override the `Notify` member function .

The `TPoolNotifier` subclass's `Notify` member function has the option of freeing up memory before resorting to growing the pool. If the `Notify` member function cannot free enough memory then it must schedule a `TGrowOperation` if it is called at interrupt time (you can call `AtInterruptLevel` to check on whether an interrupt is in progress) or just immediately grow the pool otherwise. The recommended way of doing this is to make the `TPoolNotifier` subclass' `Notify` member function call `TPoolNotifer::Notify` directly if the notifier is not going to free up any memory.

See also

`TMemoryPool`
`TGrowOperation`
`TTaskScheduler`

TPoolNotifierExample on the *ASLM Examples* disk

TPriorityLink

The `TPriorityLink` class implements a link object which can be placed on a linked list, and can hold a timer or priority value.

The `TPriorityLink` class has the following inheritance:

```
TLink  -->  TPriorityLink
```

Description

The `TPriorityLink` class is a `TLink` subclass that is designed primarily for use with the `TPriorityList` (described later in this chapter).

The `TPriorityLink` class adds an `fPriority` field to the `TLink` class, and adds two member functions for accessing the field: `SetPriority` and `GetPriority`. When a link is added to the list, the priority that is placed in the `fPriority` field determines where in the list the link will go.

Priorities used in the `fPriority` field are always unsigned long data types, three of which are predefined: `kNormalPriority`, `kHighestPriority`, and `kLowestPriority`. (The higher priority a link has, the lower is the value placed in the `fPriority` field.) To lower a link's priority, you can add the value `kToLowerPriority` to the value in the `fPriority` field.

Declarations

```
#define kNormalPriority       (((unsigned long)-1L) >> 1)

#define kHighestPriority      0
#define kLowestPriority       ((unsigned long)-1L)
#define kToLowerPriority      1

               TPriorityLink(BooleanParm);
               TPriorityLink(void* value);
               TPriorityLink(TLink* link, void* value);
               TPriorityLink();

void           SetPriority(unsigned long);
unsigned long  GetPriority() const;
```

Member functions

SetPriority
GetPriority

The `SetPriority` and `GetPriority` member functions access the `fPriority` field.

See also

`TPriorityList`
`TLink`

TPriorityList

The TPriorityList class keeps lists sorted in order of priority.

The TPriorityList class has the following inheritance:

```
TDynamic  -->  TCollection  -->  TSimpleList  -->
                                                TPriorityList
```

Description

The TPriorityList class is a subclass of TSimpleList and is maintained by TPriorityLink.

When a link is added to the list, the priority that is set for the link determines where in the list the link will go. The TPriorityList subclass adds two member functions to the TSimpleList subclasses: AddPrioritized and AddLink. The TPriorityList class does not have its own TIterator class, since the TListIterator class also works for TPriorityLists.

IMPORTANT TLink objects cannot be used on a TPriority list; only TPriorityLink objects can be used on a TPriorityList.

Declarations

```
#define kTPriorityListID "!$plst,1.1"

                         TPriorityList();
                         TPriorityList(TMemoryPool*);
                         TPriorityList(TPriorityList*);
virtual                  ~ TPriorityList();

// TLinkedList overrides

virtual   OSErr          AddFirst(void*);
virtual   OSErr          AddLast(void*);
virtual   void           AddLinkFirst(TLink*);
virtual   void           AddLinkLast(TLink*);

// New member functions

virtual   OSErr  AddPrioritized(void*, unsigned long pri);
virtual   void   AddLink(TPriorityLink*);
```

Member functions

`AddFirst`
`AddLinkFirst`
`AddLast`
`AddLinkLast`

You can use any of the `TSimpleList` member functions to add objects to a list. The `Add` member function adds the object to the list with `kLowestPriority`. Both `AddFirst` and `AddLinkFirst` set the priority of the link to `kHighestPriority`. Both `AddLast` and `AddLinkLast` set the priority of the link to `kLowestPriority`.

`AddLink`
`AddPrioritized`

The only `TPriorityList` member functions that are not in the `TSimpleList` class are `AddLink` and `AddPrioritized`. You can call `AddLink` to add a link that already has its priority set, and you can call `AddPrioritized` to add an object with the priority that you pass to `AddPrioritized`.

See also

`TCollection`
`TSimpleList`

TPriorityListExample on the *ASLM Examples* disk

TPriorityScheduler

The `TPriorityScheduler` class implements a scheduler that lets you serialize tasks by establishing their priorities.

The `TPriorityScheduler` class has the following inheritance:

```
TDynamic  -->  TScheduler  -->  TPriorityScheduler
```

Description

The `TPriorityScheduler` class allows you to schedule tasks to be processed in order of priority. The priority of the operation is set by calling the operation's `SetPriority` member function. The priority of an operation must be set before it is scheduled.

The `TPriorityScheduler`'s constructor takes an `ifAutoRun` parameter which allows you to force the scheduler to be run automatically as long it has any operations scheduled. See the description of the `Run` member function below for information on the `autorun` option.

Declarations

```
#define kTPrioritySchedulerID       "!$prsk,1.1"


                  TPriorityScheduler();   //autoRun default to false
                  TPriorityScheduler(BooleanParm ifAutoRun);
virtual           ~ TPriorityScheduler();

virtual     Boolean     IsValid() const;

virtual     Boolean     Remove(TOperation*);
virtual     TOperation* Remove(const TMatchObject&);
virtual     TOperation* RemoveNext();

virtual     Boolean     IsEmpty() const;
virtual     void        Run();
virtual     void        Schedule(TOperation*);
virtual     void        SetAutoRun(BooleanParm);
```

Member functions

IsEmpty

The `IsEmpty` member function checks to see if all scheduled operations have been executed.

IsValid

The `IsValid` member function returns `true` if the object was initialized properly after it was created. You should call `IsValid` after constructing a `TPriorityScheduler` object and delete the object if it returns `false`.

Remove

See `TScheduler::Remove` for information on the `Remove` member function.

RemoveNext

See `TScheduler::RemoveNext` for information on the `RemoveNext` member function.

Run

Ordinarily, you must explicitly call the scheduler's `Run` member function to process the operations on the scheduler. However, if you construct a `TPriorityScheduler` object by passing a `true` value in the `ifAutoRun` parameter, you do not have to call the scheduler's `Run` member function. This is because `Schedule` will automatically call `Run` if the scheduler is set up for autorun mode and it is not currently running. If the scheduler is running, it will finish processing the current operation and then continue processing operations left on the scheduler including those that were added while it was processing the operation that just finished processing. The `autorun` option can be useful when you schedule operations at interrupt time.

When in non-autorun mode, anything scheduled after `Run` is called will not be processed until `Run` is called again.

Schedule

The `Schedule` member function schedules a `TOperation`. For information on when the operation will be processed, see the description of the `Run` member function above.

SetAutoRun

The `SetAutoRun` member function sets the `autorun` option, which can be useful when you schedule operations at interrupt time. For more information, see the description of the `Run` member function above.

See also

`TOperation`
`TScheduler`

TPrioritySchedulerExample on the *ASLM Examples* disk

TProcHashObject

This is a `THashObject` subclass that uses a C procedure to hash objects.

The `TProcHashObject` class has the following inheritance:

```
TDynamic  -->  THashObject  -->  TProcHashObject
```

Description

The `TProcHashObject` class allows you to use `THashObject` class without having to subclass it. To use the `TProcHashObject` class, you can either pass the `HashProc` to the constructor or set the `HashProc` by calling `SetHashProc`. If the `HashProc` is `NULL`, the `Hash` member function will simply return the object pointer passed to it. Otherwise it will call the `HashProc.`

Declarations

```
typedef unsigned long         (*HashProc)(const void*);

#define kTProcHashObjectID "!$phob,1.1"

                         TProcHashObject(HashProc);
virtual                  ~ TProcHashObject();

virtual unsigned long    Hash(const void*) const;

     void                SetHashProc(HashProc);
     HashProc            GetHashProc() const;
```

Member functions

Hash

If the `HashProc` is `NULL`, the `Hash` member function will simply return the object pointer passed to it. Otherwise it will call the `HashProc.`

SetHashProc
GetHashProc

The `SetHashProc` and `GetHashProc` member functions set and get the object's `HashProc` .

See also

`THashObject`

TProcMatchObject

This is a TMatchObject subclass that takes a reference pointer and pointers to C functions to do the matching/hashing job.

TProcMatchObject has the following inheritance:

```
TDynamic  -->  TMatchObject  -->  TProcMatchObject
```

Description

The TProcMatchObject class allows you to use the TMatchObject class without having to subclass it. To use the TProcMatchObject class, you must pass to the constructor pointers to the HashProc, CompareProc, and IsEqualProc functions or set these functions later on by calling SetHashProc, SetCompareProc, and SetIsEqualProc. If any of these functions are set to NULL, the default functionality is used. The default action of Hash is to return the object pointer passed to it. For IsEqual, the default action is to call Compare, and for Compare the default is to compare object pointers.

Declarations

```
typedef unsigned long    (*HashProc)(const void*);
typedef Boolean          (*IsEqualProc)(const void* ref,
                         const void* toComp);
typedef int              (*CompareProc)(const void* ref,
                         const void* toComp);

#define kTProcMatchObjectID "!$pmob,1.1"

          TProcMatchObject(const void* ref, HashProc = 0,
                     CompareProc = 0, IsEqualProc = 0);
virtual                  ~ TProcMatchObject();

virtual   unsigned long  Hash() const;
virtual   short          Compare(const void*) const;
virtual   Boolean        IsEqual(const void*) const;

          void           SetReferencePointer(const void*);
          void           SetHashProc(HashProc);
          void           SetCompareProc(CompareProc);
          void           SetIsEqualProc(IsEqualProc);

const     void*          GetReferencePointer() const;
          HashProc       GetHashProc() const;
          CompareProc    GetCompareProc() const;
          IsEqualProc    GetIsEqualProc() const;
```

Continued on following page ▶

Member functions

`Compare`

If the `CompareProc` is `NULL`, the `Compare` member function will simply compare object pointers. Otherwise it will call the `CompareProc.`

`Hash`

If the `HashProc` is `NULL`, the `Hash` member function will simply return the object pointer passed to it. Otherwise it will call the `HashProc.`

`IsEqual`

If the `IsEqualProc` is `NULL`, the `IsEqual` member function will simply call `Compare`. Otherwise it will call the `IsEqualProc.`

`SetReferencePointer` `GetReferencePointer`

A `TProcMatchObject` object has a reference pointer (`refPtr`) that is attached by the creator of the match object. The `GetReferencePointer` function returns this pointer as a `void*`. The `SetReferencePointer` function sets the pointer. This pointer can be used for anything the user wants to use it for.

`SetHashProc` `GetHashProc`

The `SetHashProc` and `GetHashProc` member functions set and get the `HashProc` to be called when the object is hashed.

`SetCompareProc` `GetCompareProc`

The `SetCompareProc` and `GetCompareProc` member functions set and get the `CompareProc` to be called when the object is compared.

`SetEqualProc` `GetEqualProc`

The `SetEqualProc` and `GetEqualProc` member functions set and get the `IsEqualProc` to be called when the object is tested for equality.

See also `TMatchObject`

TProcNotifier

The TProcNotifier class is the base class for notifiers that call a C procedure for notification.

The TProcNotifier class has the following inheritance:

```
TDynamic  -->  TNotifier  -->  TProcNotifier
```

Description

The TProcNotifier class uses a pointer to a callback function and a reference pointer—an operation that works more like a traditional callback, although it is delivered via an object.

If the reference pointer is left NULL, it is replaced with a pointer to the TProcNotifier object itself. It is initially set by passing it to the TProcNotifier constructor in the refPtr parameter.

Declarations

```
#define kTProcNotifierID "!$pnot,1.1"


          TProcNotifier(NotifyProc, void* refPtr = NULL);
          TProcNotifier(const TProcNotifier&);
virtual   ~ TProcNotifier();

virtual   void Notify(EventCode, OSErrParm = kNoError,
                    void* notifyData = NULL);
```

Member functions

Notify

The Notify member function sets the global world to the global world stored with the TProcNotifier and sets the current client to the client that owns the global world. It then calls the NotifyProc that was set up when the TProcNotifier object was created. The notifyData parameter contains the reference pointer that was set up when the notifier was constructed.

See also

TNotifier

TRequestToken

The TRequestToken class keeps track of an outstanding (unfilled) request for a token.

The TRequestToken class has the following inheritance:

```
TDynamic  -->  TMatchObject  -->  TToken  -->
TRequestToken
```

Description

The TRequestToken class is the context for a request for a registered token while the request is outstanding and delivers the requested token when the request is satisfied. A TRequestToken object is a registered token itself while a request is pending. It remains registered until it is unregistered or deleted. If the request has been satisfied, the object associated with the TRequestToken is the token that was requested. Until this point, the request token has no object associated with it (GetObject will return NULL).

For more information on TRequestToken and object arbitration, see "Object Arbitration Classes" in Chapter 8, "ASLM Utility Class Categories." The descriptions of the member functions below assume that you have already read this section and understand how object arbitration works. For details on the other object arbitration classes, see "TArbitrator," "TNotifier," "TMethodNotifier," "TProcNotifier," and "TToken," in this chapter.

Declarations

```
#define kTRequestTokenID            "!$rqtk,1.1"

virtual                             ~ TRequestToken();

// TMatchObject member functions

virtual     Boolean                 IsEqual(const void*) const;
virtual     unsigned long           Hash() const;

// new member functions

virtual     Boolean                 Give(TToken* theToken);
virtual     TToken*                 Exchange();
virtual     void                    RequestAgain();

virtual     TokenRequestType        GetRequestType() const;
virtual     void                    SetRequestType(TokenRequestType);

            Boolean                 IsTokenRegistered() const;
```

Member functions

Exchange

The `Exchange` member function polls for the requested token. `Exchange` returns `NULL` for as long as the request is not satisfied. When the request is satisfied, `Exchange` deletes the `TRequestToken` object and returns the requested token.

Give

The `Give` member function is used to give up the specified token to the `TRequestToken` object. If `TRequestToken` has a `TNotifier` object, `TRequestToken` calls its `Notify` member function to notify the requester that the request has been satisfied. The `notifyData` parameter in the `Notify` call is a `TTokenNotification` object containing the request token and the requested token.

Hash

The `Hash` member function is used internally by `TArbitrator` in order to determine which hash bucket the requested token can be found in (the `TArbitrator` uses a `THashList` to store registered tokens).

IsEqual

The `IsEqual` member function is used internally by `TArbitrator` in order to determine which tokens match the token that the request token is requesting.

IsTokenRegistered

The `IsTokenRegistered` member function is used to determine whether the token that was requested is actually registered already. Even if the token is not registered yet, the request token will still be notified when the token is registered.

RequestAgain

If you want to use a `TRequestToken` for multiple requests of the same token id, you can call `TToken::GetObject` to poll for the requested token. If `GetObject` returns a token, you can call `RequestAgain` to request another token using the same `TRequestToken`. You can use this technique to get several tokens of the same type—but do not release any of them until the process is complete. If a token is released before you call `RequestAgain`, the same token is returned.

Continued on following page ▶

`SetRequestType`
`GetRequestType`

`SetRequestType` is used to set the request type of the request token and `GetRequestType` will return the request type of the request token. There is normally no reason to change a request token's request type since request tokens are created for you when you when you request a token from a `TArbitrator` and are deleted when you call `Exchange`.

See also

`TNotifier`
`TMethodNotifier`
`TProcNotifier`
`TToken`
`TTokenNotification`

"Object arbitration classes" in Chapter 8, "ASLM Utility Class Categories."

TArbitratorExample1, TArbitratorExample2, and TArbitratorExample3 examples on the ASLM Examples disk.

TSCDynamic

The `TSCDynamic` class provides the same capabilities as `TStdDynamic`. However, it is only used for Symantec C++ implementations.

Description

The `TSCDynamic` class is the base class for shared library classes with a set of common capabilities.

For more information on `TSCDynamic` and `TStdDynamic`, see “The TDynamic Family of Base Classes” in Chapter 6, “Using the ASLM.”

Declarations

```
#define kTSCDynamicID "!$scdy,1.1"

virtual                 ~TSCDynamic();

      void*             operator new(size_t size, TMemoryPool*);
                              // from specified pool
      void*             operator new(size_t);   // from default pool
      void              operator delete(void* obj, size_t)
                              { SLMDeleteOperator(obj); }

const TClassID&         GetObjectsClassID() const;
const TClassID&         GetObjectsParentClassID() const;
      size_t            GetObjectsSize() const;
      TLibrary*         GetObjectsLocalLibrary() const;
      TLibraryFile*     GetObjectsLocalLibraryFile() const;
      TStandardPool*    GetObjectsLocalPool() const;
      void              SetObjectsLocalPool(TStandardPool*) const;

virtual     Boolean     _cdecl IsValid() const;

virtual     OSErr       _cdecl Inflate(TFormattedStream&);
virtual     OSErr       _cdecl Flatten(TFormattedStream&) const;
virtual     TSCDynamic* _cdecl Clone(TStandardPool*) const;

virtual     char*       _cdecl GetVerboseName(char*) const;
virtual     void        _cdecl Dump() const;

            void        Trace(char *formatStr, ...) const;
virtual     Boolean     _cdecl TraceControl(TraceControlType) const;
      Boolean           IsTraceOn() const;
      Boolean           TraceOn() const;
      Boolean           TraceOff() const;

      Boolean           IsDerivedFrom(const TClassID&) const;
```

Continued on following page ▶

Member functions See “TDynamic” for information on the member functions of `TSCDynamic`.

TScheduler

The `TScheduler` objects are used to schedule `TOperations` for later execution and to control when the `TOperations` are executed (processed).

The `TScheduler` class has the following inheritance:

```
TDynamic  -->  TScheduler
```

Description

The `TScheduler` class, the base class for all schedulers, is an abstract class that you must inherit from in order to use. The ASLM provides a number of schedulers for you including `TTaskScheduler`, `TInterruptScheduler`, `TTimeScheduler`, `TThreadScheduler`, `TSerialScheduler`, and `TPriorityScheduler`. Some general information about schedulers can be found in "Process Management Classes" in Chapter 8, "ASLM Utility Class Categories."

For instructions on setting up a global world for an operation and setting a scheduler's global world, see "TOperation" earlier in this chapter.

Declarations

```
#define kTSchedulerID         "!$sked,1.1"

virtual                 ~ TScheduler();

virtual   Boolean       Remove(TOperation*) = 0;
virtual   TOperation*   Remove(const TMatchObject&)   = 0;
virtual   TOperation*   RemoveNext()                  = 0;
virtual   Boolean       IsEmpty() const               = 0;

virtual   void          Schedule(TOperation*)         = 0;
virtual   void          Run()                         = 0;

          Boolean       IsSchedulerWorldValid() const;
          GlobalWorld   GetSchedulerWorld() const;
          void          SetSchedulerWorld(GlobalWorld);
```

Member functions

IsEmpty

The `IsEmpty` member function checks to see if all scheduled operations have been executed.

Continued on following page ▶

IsSchedulerWorldValid
GetSchedulerWorld
SetSchedulerWorld

A scheduler has a global world associated with it that is used to set up the global world for operations before they are processed. See “TOperation” earlier in this chapter for more information on setting up the global world for operations.

The `IsSchedulerWorldValid` member function returns `true` if the scheduler’s world is set to a valid one (anything but `kInvalidWorld`). By default the scheduler’s global world is set to `kInvalidWorld`.

The `GetSchedulerWorld` and `SetSchedulerWorld` member functions are used to get and set the scheduler’s global world.

Remove

When a `TOperation` is scheduled, you can remove it by calling the `Remove` member function. Calling `Remove(TOperation*)` returns `true` if the operation is removed. `Remove(const TMatchObject&)` returns the object that was removed, otherwise it returns `NULL`

RemoveNext

`Remove` removes only the first operation that matches the operation or `TMatchObject` object that is passed to it. To remove the next `TOperation` object to be processed, you can call `RemoveNext`. `RemoveNext` will return `NULL` if there are no more operations on the scheduler.

Schedule
Run

The most important `TScheduler` member functions are `Schedule`, which schedules a `TOperation`, and `Run`, which processes all scheduled `TOperations`. Processing includes removing `TOperation` objects from the `TScheduler` object as they are processed. If you want to reschedule a `TOperation` object that has just been processed, you must reschedule it yourself.

To reschedule a `TOperation` object that has just been processed, call `Schedule`. The newly scheduled `TOperation` will not be processed again until the `TScheduler` object's `Run` member function is called again. The creator of the `TScheduler` object is responsible for determining when the object's `Run` member function is called, although some `TSchedule` subclasses have an autoRun feature that causes them to be run every time you try to schedule a `TOperation` object. Also, some schedulers determine for themselves when they should be run. For example, the `TTaskScheduler` processes its operations automatically at System Task time.

See also

`TOperation`
`TTaskScheduler`
`TTInterruptScheduler`
`TTimeScheduler`
`TThreadScheduler`
`TSerialScheduler`
`TPriorityScheduler`

TSeconds

This `TTime` subclass is used to specify an initial time value in seconds—that is, it provides a constructor that takes a time value in seconds.

The `TSeconds` class has the following inheritance:

```
TDynamic  -->  TMatchObject  -->  TDoubleLong  -->
                                     TTime  -->  TSeconds
```

Description For additional information, see "TTime" later in this chapter.

Declarations

```
#define kTSecondsID "slm:supp$secs,1.1"

                         TSeconds();
                         TSeconds(unsigned long secs);
                         ~ TSeconds();

          operator       unsigned long() const;
virtual   double         ConvertToDouble() const;
                         operator double() const;
```

Member functions

`operator unsigned long`

The `operator unsigned long` member function returns the number of seconds in an `unsigned long.`

`ConvertToDouble`

The `ConvertToDouble` member function converts the time to a `double` containing the number of seconds.

`operator double`

The `operator double` member function returns the number of seconds in a `double` by calling `ConvertToDouble`.

See also TTimeExample on the *ASLM Examples* disk

TSerialScheduler

The `TSerialScheduler` class is a `TPriorityScheduler` subclass that ensures FIFO (first in, first out) processing of the `TOperation` class.

The `TSerialScheduler` class has the following inheritance:

```
TDynamic  -->  TScheduler  -->  TPriorityScheduler  -->
                                        TSerialScheduler
```

Description

The `TSerialScheduler` class works like a `TPriorityScheduler`, but always sets the operation's priority to `kNormalPriority` before scheduling it.

Declarations

```
#define kTSerialSchedulerID   "!$srsk,1.1"

          TSerialScheduler(); // autoRun default to false
          TSerialScheduler(BooleanParm ifAutoRun);
virtual   ~ TSerialScheduler();

virtual   void Schedule(TOperation*);
```

Member functions

Schedule

The `Schedule` member function schedules a `TOperation`. It overrides `TPriorityScheduler::Schedule` to set the operation's priority to `kNormalPriority`. Otherwise it behaves the same as `TPriorityScheduler::Schedule`.

See also

`TScheduler`
`TPriorityScheduler`

TSerialSchedulerExample on the *ASLM Examples* disk

TSimpleDynamic

The TSimpleDynamic class forces the v-table first.

The TSimpleDynamic class has no parent class.

Description

The TSimpleDynamic class is base class for shared-library classes that has no virtual functions. This class is not a shared class, since it is intended to be a class that just forces the v-table to be at the front of the object.

The TSimpleDynamic class inherits from SingleObject and only has one virtual member function: its destructor. This feature gives TSimpleDynamic a small, simple v-table.

For more information on TSimpleDynamic, see “The TDynamic Family of Base Classes” in Chapter 6, “Using the ASLM.”

Declarations

```
virtual          ~ TSimpleDynamic();

void*            operator new(size_t size, TMemoryPool*);
                       // from specified pool
void*            operator new(size_t);   // from default pool
void             operator delete(void* obj, size_t)
                       { SLMDeleteOperator(obj); }

const TClassID&        GetObjectsClassID() const;
const TClassID&        GetObjectsParentClassID() const;
size_t                 GetObjectsSize() const;
TLibrary*              GetObjectsLocalLibrary() const;
TLibraryFile*          GetObjectsLocalLibraryFile() const;
TStandardPool*         GetObjectsLocalPool() const;
void                   SetObjectsLocalPool(TStandardPool*) const;

Boolean                IsDerivedFrom(const TClassID&) const;
```

Member functions

See “TDynamic” earlier in this chapter for a description of the TSimpleDynamic member functions.

TSimpleList

The `TSimpleList` class is a `TCollection` subclass that implements a linked list that can have objects added at the front or the back.

The `TSimpleList` class has the following inheritance:

```
TDynamic  -->  TCollection  -->  TSimpleList
```

Description

The `TSimpleList` class is used to maintain a list of objects. It uses `TLinks` to maintain the list and is a very efficient collection class when objects do not need to be looked up or removed from the middle of the list.

The ASLM provides two `TSimpleList` subclasses. The `TLinkedList` provides a couple of useful additional methods and the `TPriorityList` is used to maintain a list of objects sorted by priority.

A `TListIterator` class is provided to iterate through the linked list. The `TSimpleList` class provides some member functions besides those that belong to the `TCollection` class. The names of the member functions provided by `TSimpleList` are intuitive and largely self-explanatory.

The constructor that takes a `TMemoryPool*` parameter is used to specify the pool (called the link pool) to be used when `TLink` objects need to be allocated for objects added to the pool. It is recommended that you use a `TChunkyPool` for the link pool since it is more efficient than a `TStandardPool` at allocating blocks of memory of the same size. `TLinks` are allocated out of the link pool automatically whenever an object (as opposed to a link) is added to the list. They are also deleted automatically whenever an object is removed or deleted from the list.

When you remove or delete objects from a linked list, the `TLink` objects are deleted only if the link pool has been set. However, when you remove links (by calling `RemoveFirstLink` or `RemoveLastLink`), the `TLink` is returned and not deleted, and you are responsible for deleting them.

IMPORTANT It is not safe to set the link pool and call `AddLinkFirst` or `AddLinkLast` unless you have explicitly allocated the link from a pool yourself. Adding `TLink` objects that are stack objects or class data members is not allowed if the link pool has been set.

Continued on following page ▶

Declarations

```
#define kTSimpleListID "!$slst,1.1"

                         TSimpleList();
                         TSimpleList(TMemoryPool*);
                         TSimpleList(TSimpleList*);
virtual                  ~ TSimpleList();

// TCollection overrides

virtual   TIterator*     CreateIterator(TStandardPool*);

virtual   void*          Remove(const TMatchObject&);
virtual   void*          Member(const TMatchObject&);
virtual   Boolean        Remove(void*);
virtual   Boolean        Member(const void*);

     // New members

virtual   TLink*         MemberLink(const void*);
virtual   TLink*         MemberLink(const TMatchObject&);
virtual   TLink*         RemoveLink(void*);
virtual   TLink*         RemoveLink(const TMatchObject&);

virtual   TLink*         FirstLink() const;
virtual   TLink*         LastLink() const;
virtual   TLink*         RemoveFirstLink();
virtual   TLink*         RemoveLastLink();
virtual   void           AddLinkFirst(TLink*);
virtual   void           AddLinkLast(TLink*);

virtual   void*          First() const;
virtual   void*          Last() const;
virtual   void*          RemoveFirst();
virtual   void*          RemoveLast();
virtual   OSErr          AddFirst(void*);
virtual   OSErr          AddLast(void*);
          void           SetLinkPool(TMemoryPool*);
          TMemoryPool*   GetLinkPool() const;
```

Member functions

Add
AddUnique

The `Add` and `AddUnique` member functions are described in "TCollection" earlier in this chapter.

AddFirst
AddLast

The `AddFirst` member function adds the object to the beginning of the list, and the `AddLast` member function adds the object to the end of the list. The `AddFirst` and `AddLast` member functions return error codes other than `kNoError` if they fail to add the object to the list.

AddLinkFirst
AddLinkLast

The `AddLinkFirst` member function adds the link to the beginning of the list, and the `AddLinkLast` member function adds the link to the end of the list. If you do not set the link pool, the only `TSimpleList` member functions you can use to add links to the list are `AddLinkFirst` and `AddLinkLast`. The `AddLinkFirst` and `AddLinkLast` member functions can be useful when an object you want to add to the linked list has a `TLink` field that points to itself.

CreateIterator

The `CreateIterator` member function returns an iterator for a `TSimpleList` object. For additional information, see “TListIterator” earlier in this chapter.

First
Last

The `First` member function returns the first object in the list, and the `Last` member function returns the last object in the list.

FirstLink
LastLink

The `FirstLink` member function returns the link for the first object in the list, and the `LastLink` member function returns the link for the last object in the list.

GetLinkPool
SetLinkPool

The `GetLinkPool` member function returns the link pool for the list. The `SetLinkPool` member function sets the link pool for the list.

Continued on following page ▶

The `TLink` objects are used to maintain linked lists. The ASLM allocates `TLink` objects when you call `Add`, `AddUnique`, `AddFirst`, or `AddLast`. However, before you call any of these member functions, you must set the `TMemoryPool` from which you want your `TLink` objects to allocate memory. You can do this by passing the `TMemoryPool` object to the constructor or by calling `SetLinkPool`.

`Member`

The `Member` member function is described in “TCollection” earlier in this chapter.

`MemberLink`

The `MemberLink` member function returns the `TLink` of the object passed if it is in the collection. It searches for the object in the same manner as `Member.`

`RemoveFirstLink`
`RemoveLastLink`

The `RemoveFirstLink` member function removes the first link from the collection, and the `RemoveLastLink` member function removes the last link from the collection.

`RemoveFirst`
`RemoveLast`

The `RemoveFirst` member function removes the first object from the list, and the `RemoveLast` member function removes the last object from the list.

See also `TCollection`

TSimpleListExample on the *ASLM Examples* disk

TSimpleRandom

The `TSimpleRandom` class returns a random number computed with 64-bit arithmetic.

The `TSimpleRandom` class has the following inheritance:

```
TDynamic  -->  TFastRandom  -->   TSimpleRandom
```

Description

The `TSimpleRandom` class generates better random numbers over a 32-bit range than `TFastRandom` does.

Declarations

```
#define kTSimpleRandomID "slm:supp$srnd,1.1"


const unsigned long     kMaxSimpleRandom = 2145740624;

                        TSimpleRandom();
                        TSimpleRandom(unsigned long seed);
                        TSimpleRandom(unsigned long im,
                              unsigned long ia, unsigned long ic);
virtual                 ~ TSimpleRandom();

virtual unsigned long   GetRandom();
virtual unsigned long   GetRandomNumber(unsigned long lo,
                              unsigned long hi);
```

Member functions

GetRandom
GetRandomNumber

The `GetRandom` member function returns a random number from 0 to `kMaxSimpleRandom`, inclusive. The `GetRandomNumber` member function returns the a random number from `lo` to `hi`, inclusive. You should normally use `GetRandomNumber` instead of `GetRandom`.

TSimpleRandom

The `TSimpleRandom` member function creates an object, using the current time stamp as the seed.

The `TSimpleRandom(unsigned long seed)` function creates the object using the value of *`seed`* as the seed

The `TSimpleRandom(unsigned long im, unsigned long ia, unsigned long ic)` function creates the object using the current time stamp as the seed, and the parameters as the random number generator using a standard congruency generator:

```
seed = (seed*ia + ic) % im;
```

TStandardPool

The TStandardPool class provides a general purpose, interrupt-safe memory allocator.

The TStandardPool class has the following inheritance:

```
TDynamic  -->  TMemoryPool  -->  TStandardPool
```

Description

The TStandardPool class is a TMemoryPool subclass that implements a general purpose, interrupt-safe memory allocator. Besides being interrupt-safe, it is also much faster than other common memory allocators such as the Macintosh Memory Manager and C calloc and malloc functions. See "TMemoryPool" earlier in this chapter for more information on memory pools.

The TStandardPool class provides a constant named kStandardPoolChunkOverhead that can help you determine the amount of overhead that each chunk allocated from a pool will require. Use this constant to help you decide how big a pool you will need.

The following code fragment shows how to create a standard pool, add some additional memory to it, and then destroy it (In this example, the system zone is used for the memory pool.):

```
TStandardPool* myPool = new (kMyPoolSize,kSystemZone)
                                   TStandardPool;
myPool->AddMemoryToPool(kMyExtraMemorySize);
delete myPool;
```

The following is an example of how the constant kStandardPoolChunkOverhead can be used. If you expect to allocate 100 blocks from a pool, and you estimate that the blocks will occupy no more than 5000 bytes of memory, you can create the pool by writing these two statements:

```
size_t poolsize = 100 * kStandardPoolChunkOverhead + 5000;
TStandardPool* myPool = new (poolsize, kSystemZone)
TStandardPool;
```

When memory is allocated, block sizes are always rounded up to be a multiple of 8. This needs to be taken into account when determining how large a pool you need.

Declarations

```
#define kStandardPoolChunkOverhead          12

#define kTStandardPoolID "!$stdp,1.1"

                              TStandardPool();
virtual                       ~ TStandardPool();

virtual Boolean               IsValid() const;

// TMemoryPool Overrides

virtual   void*               Allocate(size_t);
virtual   void*               Reallocate(void*, size_t);
virtual   void                Free(void*);
virtual   size_t              GetSize(void*) const;
virtual   Boolean             CheckPool() const;
virtual   size_t              GetLargestBlockSize() const;
```

Member functions

The member functions are described in “TMemoryPool” earlier in this chapter.

IsValid

The `IsValid` member function returns `false` if the pool is corrupt or was not created properly. It should not be used in this case.

TStdDynamic

The `TStdDynamic` class is the base class for shared-library classes with a set of common capabilities.

The `TStdDynamic` class has no parent class.

Description

The `TStdDynamic` class is similar to `TDynamic`, except it does not inherit from `SingleObject`. It has the same functionality as `TDynamic`, except it cannot be registered with the Inspector application. It is useful when you do not want to inherit from `SingleObject`, but you do want most of the extra methods that `TDynamic` provides. It also forces the v-table to be first.

For more information on `TStdDynamic`, see "The TDynamic Family of Base Classes" in Chapter 6, "Using the ASLM."

Declarations

```
#define kTStdDynamicID "!$sdyn,1.1"

virtual                 ~ TStdDynamic();

      void*             operator new(size_t size, TMemoryPool*);
                              // from specified pool
      void*             operator new(size_t);   // from default pool
      void              operator delete(void* obj, size_t)
                              { SLMDeleteOperator(obj); }

const TClassID&         GetObjectsClassID() const;
      size_t            GetObjectsSize() const;
      TLibrary*         GetObjectsLocalLibrary() const;
      TLibraryFile*     GetObjectsLocalLibraryFile() const;
      TStandardPool*    GetObjectsLocalPool() const;
      void              SetObjectsLocalPool(TStandardPool*) const;

virtual     Boolean     IsValid() const;

virtual     OSErr       Inflate(TFormattedStream&);
virtual     OSErr       Flatten(TFormattedStream&) const;
virtual     TDynamic*   Clone(TStandardPool*) const;

virtual     char*       GetVerboseName(char*) const;
virtual     void        Dump() const;
```

```
            void        Trace(char *formatStr, ...) const;
virtual     Boolean     TraceControl(TraceControlType) const;
      Boolean           IsTraceOn() const;
      Boolean           TraceOn() const;
      Boolean           TraceOff() const;

      Boolean           IsDerivedFrom(const TClassID&) const;
```

Member functions

See “TDynamic” earlier in this chapter for a description of the `TStdDynamic` member functions.

TStdSimpleDynamic

The `TStdSimpleDynamic` class is the base class for shared-library classes that have no virtual functions.

The `TStdSimpleDynamic` class has no parent class.

Description

This class is *not* shared, since it is a class that just forces the v-table to be at the front of the object.

The `TStdSimpleDynamic` class works like `TSimpleDynamic`, except that it does not inherit from `SingleObject`. The `TStdSimpleDynamic` class, like `TSimpleDynamic`, has a small v-table, but it is not simple because it does not inherit from `SingleObject`. It is useful if you want some of the non-virtual member functions that `TDynamic` provides that give you meta information about the object.

For more information on `TStdSimpleDynamic`, see "The TDynamic Family of Base Classes" in Chapter 6, "Using the ASLM."

Declarations

```
virtual                             ~ TStdSimpleDynamic();


      void* operator new(size_t size, TMemoryPool*);
                  //from specified pool
      void* operator new(size_t);               // from default pool
      void  operator delete(void* obj, size_t)
                  { SLMDeleteOperator(obj); }

      const TClassID&   GetObjectsClassID() const;
      size_t            GetObjectsSize() const;
      TLibrary*         GetObjectsLocalLibrary() const;
      TLibraryFile*     GetObjectsLocalLibraryFile() const;
      TStandardPool*    GetObjectsLocalPool() const;
      void              SetObjectsLocalPool(TStandardPool*) const;

      Boolean           IsDerivedFrom(const TClassID&) const;
```

Member functions

See "TDynamic" earlier in this chapter for a complete description of the `TStdSimpleDynamic` member functions.

TStopwatch

The `TStopwatch` class is used to determine the time that has elapsed since the `TStopwatch` object was initialized.

The `TStopwatch` class has the following inheritance:

```
TDynamic  -->  TMatchObject  -->  TDoubleLong  -->
                    TTime  -->  TTimeStamp  -->  TStopwatch
```

Description

The `TStopwatch` class is a `TTimeStamp` subclass that remembers a time stamp when it is initialized and compares that time stamp to a new time stamp that is taken each time one of the “elapsed” routines, such as `ElapsedSeconds`, is called. A `TStopwatch` object is initialized when it is constructed and whenever `Reset` is called.

Declarations

```
#define kTStopwatchID "slm:supp$stpw,1.1"


                              TStopwatch();
virtual                       ~ TStopwatch();

virtual void                  Reset();

virtual unsigned long         ElapsedMicroseconds() const;
virtual unsigned long         ElapsedMilliseconds() const;
virtual unsigned long         ElapsedSeconds() const;
```

Member functions

ElapsedMicroseconds
ElapsedMilliseconds
ElapsedSeconds

The `ElapsedMicroseconds`, `ElapsedMilliseconds`, and `ElapsedSeconds` member functions return the number of microseconds, milliseconds, or seconds that have elapsed since `TStopwatch` was created or last reset.

Reset

The `Reset` function restarts the stopwatch by setting the time stamp to the current time.

See also

`TTime`

TTimeExample on the *ASLM Examples* disk

TTaskScheduler

The `TTaskScheduler` class implements a scheduler for heavyweight tasks.

The `TTaskScheduler` class has the following inheritance:

```
TDynamic  -->  TScheduler  -->  TPriorityScheduler  -->
                                       TTaskScheduler
```

Description

The `TTaskScheduler` class is a `TPriorityScheduler` subclass that can be useful for scheduling heavyweight tasks (that is, tasks that consume a great amount of CPU time or use system resources). Specifically, `TTaskScheduler` schedules `TOperation` objects to run at System Task time.

The ASLM has a global `TTaskScheduler` class that clients can use instead of creating their own scheduler. You can access the global `TTaskScheduler` class by calling `GetGlobalTaskScheduler`. You should call the `IsValid` member function before you use a newly created `TTaskScheduler` to verify that it was initialized properly and can be used.

```
TTaskScheduler*           GetGlobalTaskScheduler();
```

For more information on schedulers, see “Process Management Classes” in Chapter 8, “ASLM Utility Class Categories,” and “TScheduler” earlier in this chapter.

Declarations

```
#define kTTaskSchedulerID           "!$task,1.1"


                              TTaskScheduler();
                              TTaskScheduler(unsigned long priority,
                                    BooleanParm runToEmpty = false);
virtual                       ~ TTaskScheduler();

virtual     void              Schedule(TOperation*);

virtual     void              SetPriority(unsigned long);
            void              SetRunToEmpty(Boolean);
```

Member functions

Schedule

The `Schedule` member function schedules a `TOperation` object. The operation will be run at the next System Task time.

`SetPriority`

The `TTaskScheduler` objects have priorities. These priorities determine the order in which each `TTaskScheduler` is processed at System Task time. The priority of a `TTaskScheduler` is either passed in the constructor or set by the `SetPriority` member function. The default priority of every `TTaskScheduler` object is `kNormalPriority`. The ASLM global task scheduler has a priority one higher than `kNormalPriority`, so it is processed first, unless you give your `TTaskScheduler` a higher priority.

`SetRunToEmpty`

The `TTaskScheduler` objects also have a flag named `runToEmpty`. If this flag is `true`, operations that are scheduled while the `TTaskScheduler` is already running are run during the current System Task time rather than waiting for the next System Task time. The `runToEmpty` flag defaults to `false`. It can be set in the constructor, or it can be set by calling `SetRunToEmpty`.

See also `TScheduler`

TSchedulerExample on the *ASLM Examples* disk

TTestTool

The TTestTool class is a class used by the MPW tool TestTool, provided with the ASLM.

The TTestTool class has the following inheritance:

```
TDynamic  -->  TTestTool
```

Description

TestTool is a tool for writing ASLM classes that can be used as test modules. The test modules are run by TestTool, an MPW tool provided with ASLM. TestTool is described in Appendix B, “ASLM Utility programs.”

Declarations

```
#define kTTestToolID kTestToolPrefix "TTestTool,1.1"

               TTestTool();
               TTestTool(TStandardPool* thePool);
virtual        ~TTestTool();

virtual void   SetPrintf(PrintfFunc);
virtual void   Printf(const char*, ...) const;
virtual void   InitTest(Boolean verbose, Boolean debug, int argc,
                                          char** argv)      = 0;
virtual void   RunTestIteration(Boolean verbose, Boolean debug) = 0;
virtual void   EndTest(Boolean verbose, Boolean debug)     = 0;

      void             SetPool(TStandardPool* thePool);
      TStandardPool*   GetPool();
```

Member functions

EndTest

The EndTest member function is called when the MPW tool TestTool has finished running the test. At this point you should clean up.

InitTest

The InitTest member function is called just before the MPW tool TestTool starts running the test. At this point, you should do most of your setting up for the test.

Printf

The Printf member function is the same as the C Printf routine, except that it works with a shared library, allowing TestTool to send text to the MPW Worksheet.

`RunTestIteration`

The `RunTestIteration` member function runs one iteration of the test.

`SetPool`

The `SetPool` member function sets the pool out of which your test should allocate memory. It is usually called by the MPW tool TestTool.

`SetPrintf`

The `SetPrintf` member function sets the routine to call for `Printf`. It is usually set by the MPW tool TestTool.

`TThreadScheduler`

The `TThreadScheduler` class is a `TPriorityScheduler` subclass that implements a lightweight “thread” scheduler.

The `TThreadScheduler` class has the following inheritance:

```
TDynamic  --> TScheduler  -->  TPriorityScheduler  -->
                                        TThreadScheduler
```

Description

In the Macintosh implementation, `TThreadScheduler` works like `TPriorityScheduler` with the autorun option on. A thread is a lightweight task that has no operating system calls. You must call the `IsValid` member function to verify that the scheduler was initialized properly and can be used.

The `Run` member function of `TThreadScheduler` is private and should never be called. It is provided so that if the Macintosh someday has real threads, or if the ASLM is ported to an operating system with threads, users of `TThreadScheduler` automatically get this functionality.

Declarations

```
#define kTThreadSchedulerID   "slm:sked$thsk,1.1"

                              TThreadScheduler();
virtual                       ~ TThreadScheduler();

virtual void                  Schedule(TOperation*);
```

Member functions

`Schedule`

The `Schedule` member function schedules the operation and calls `Run` if it is not already running. It behaves the same as `TPriorityScheduler` with the autorun option on.

See also

`TScheduler`
`TPriorityScheduler`

TTime

The `TTime` class is the base class for all time-related classes.

The `TTime` class has the following inheritance:

```
TDynamic -->  TMatchObject  -->  TDoubleLong  -->  TTime
```

Description

The ASLM provides several special data-type classes to help libraries and clients perform “time math.” The `TTime` class obtains a time value from the CPU’s time-generating system and provides all the routines for accessing that value. You can use a `TTimeStamp` to initialize a `TTime` with a time. Since `TTime` is a `TDoubleLong` subclass, you can perform all the 64-bit integer math operations on it also.

Internally, all times are stored as microseconds, and the casting operators for the `TTime` class return values converted to other time units. The `TMicroSeconds`, `TMilliSeconds`, and `TSeconds` subclasses are used to provide an initial time in microseconds, milliseconds, or seconds.

Declarations

```
#define kTTimeID "slm:supp$time,1.1"

                    TTime();
                    TTime(unsigned long microseconds);
                    TTime(const TDoubleLong&);
                    TTime(const TTime&);
virtual             ~ TTime();

          void      SetTime(const TTime&);

          void      SetMicroseconds(unsigned long);
virtual   void      SetMilliseconds(unsigned long);
virtual   void      SetSeconds(unsigned long);

          unsigned long  GetMicroseconds() const;
virtual   unsigned long  GetMilliseconds() const;
virtual   unsigned long  GetSeconds() const;
```

Member functions

`GetMicroseconds`
`GetMilliseconds`
`GetSeconds`

These three member functions get the time in microseconds, milliseconds, and seconds.

Continued on following page ▶

`SetMicroseconds`
`SetMilliseconds`
`SetSeconds`

The `SetMicroseconds`, `SetMilliseconds`, and `SetSeconds` member functions set the time in microseconds, milliseconds, and seconds.

`SetTime`

The `SetTime` member function sets the time based on the `TTime` object passed to it.

See also

`TTimeStamp`
`TStopWatch`
`TMillisconds`
`TMicroSeconds`
`TSeconds`

TTimeExample on the *ASLM Examples* disk

TTimeScheduler

The `TTimeScheduler` class implements a scheduler that processes `TOperation` objects when a requested amount of time has elapsed.

The `TTimeScheduler` class has the following inheritance:

```
TDynamic  -->  TScheduler  -->  TPriorityScheduler  -->
                                          TTimeScheduler
```

Description

You must call the `TOperation::SetTime` member function before scheduling a `TOperation` object to determine when the `TOperation` will be processed. On the Macintosh, `TTimeScheduler` is a front end to the Time Manager. Operations scheduled on a `TTimeScheduler` object execute at interrupt time.

Schedulers within schedulers

All `TTimeScheduler` objects can be combined with other kinds of schedulers to handle operations that otherwise would be processed immediately. The most common use of this capability is to provide a `TTimeScheduler` object with a `TInterruptScheduler` object. Then, when a `TTimeScheduler` operation fires, it is automatically placed on a `TInterruptScheduler` object so it can be processed at deferred task time rather than immediately at interrupt time. You can provide a `TTimeScheduler` object with another scheduler by passing it to the constructor.

Member functions for handling interrupts

The ASLM provides four member functions that can be useful when you want to remove a `TOperation` object from a `TTimeScheduler` object at interrupt time. These four member functions—`DeleteInProcessOperation`, `RerunInProcessOperation`, `Reschedule`, and `SetAutoReschedule`—give you some amount of control over the problem of removing a `TOperation` object while its `Process` routine is being called.

The `DeleteInProcessOperation`, `RerunInProcessOperation`, `Reschedule`, and `SetAutoReschedule` member functions are provided because `Remove` calls that are invoked at interrupt time can remove a `TOperation` object that is in the process of being executed. (The `Remove` member function is inherited from the `TScheduler` class.) Ordinarily, deleting or rescheduling a `TOperation` object that is in process can be disastrous.

Continued on following page ▶

When you have a client that needs to remove a `TOperation` object while an interrupt is in progress, you can call the `TOperation::WasRemovedInProcess` member function to determine if it was removed while it was in the `Process` member function.

If `WasRemovedInProcess` returns `false`, you are free to do whatever you want with the `TOperation` object. If `WasRemovedInProcess` returns `true`, you must decide what you want to do with the in-process `TOperation`. You can call either `DeleteInProcessOperation`, which informs the scheduler that it should delete the `TOperation` when it returns from the `Process` call, or you can call `RerunInProcessOperation`, which causes the scheduler to call `Process` again immediately after returning from the `Process` member function.

A `TOperation` object can call the `TOperation` member function `WasRemovedInProcess` to determine whether it has been removed. It may then modify its behavior accordingly. If a `TOperation` has been removed while it is in process, it can delete itself, provided it calls `TOperation::ClearRemovedInProcess` so that `TTimeScheduler` does not do anything with the deleted `TOperation` object.

It is important to remember, however, that the behavior of the "remover" and `TOperation` must be coordinated. If the `TOperation` deletes itself when it detects that it was removed, then you can probably call either `DeleteInProcessOperation` or `RerunInProcessOperation`, since in either case, the `TOperation` is deleted. However, if the `TOperation` must do something more complicated, then you should use `RerunInProcessOperation` to give the `TOperation` a chance to do whatever it has to do.

If a `TOperation` object detects that it was removed in process, and is about to act on it itself, then it should call the `TOperation` member function `ClearRemovedInProcess` to keep `TTimeScheduler` from either deleting the operation or rerunning it (depending on what the "remover" asked for).

If a `TOperation` object wants to delete itself when auto-rescheduling is true, and it has not been removed in process, it should ensure that its time field is 0 and then call `SetDeleteWhenDone`. This operation informs the `TTimeScheduler` that a `TOperation` object wants to be deleted. If your `TOperation` is removed before `SetDeleteWhenDone` returns to the `TTimeScheduler`, the rules governing removal in process take effect.

A `TOperation` object may also delete itself by calling the `TTimeScheduler Remove` member function and, if that succeeds, calling the `DeleteInProcessOperation` member function. But this requires that you know which `TTimeScheduler` the `TOperation` object is on. The previous technique does not require this knowledge.

A particularly tricky situation can occur when a `TOperation` is an embedded object. In this case, you want to delete the object in which the `TOperation` is embedded, not the `TOperation` itself.

The only way to do this safely is to call the `RerunInProcessOperation` member function. To do that, you can place code like this in your `Process` member function:

```
If (WasRemovedInProcess())
{
// Do your standard stuff, but be aware that you
// might come here twice.

     if (IsBeingRerun())
          {
               delete parentObject;
          }
     return;
}

// Here, do your normal thing.
```

This technique requires that the "remover" of your operation call `RerunInProcessOperation` whenever it detects that your `TOperation` object was removed in process.

The `DeleteInProcessOperation`, `RerunInProcessOperation`, and `SetAutoReschedule` member functions are described in more detail in "Member Functions" below.

Declarations

```
#define kMaxScheduledTime     ((unsigned long)-1L)

#define kTTimeSchedulerID     "slm:sked$skti,1.1"

                              TTimeScheduler();
                              TTimeScheduler(void* data);
                              TTimeScheduler(TScheduler*,
                                   unsigned long priority);
virtual                       ~ TTimeScheduler();

virtual Boolean          IsValid() const;
```

Continued on following page ▶

```
virtual Boolean          Remove(TOperation*);
virtual TOperation*      Remove(const TMatchObject&);
virtual TOperation*      RemoveNext();
virtual void             Schedule(TOperation*);
virtual Boolean          IsEmpty() const;

virtual Boolean          Reschedule(TOperation*,
                              unsigned long time);
virtual void             SetAutoReschedule(BooleanParm);

virtual Boolean   DeleteInProcessOperation(TOperation* op);
virtual Boolean   RerunInProcessOperation(TOperation* op);
```

Member functions

DeleteInProcessOperation

The `DeleteInProcessOperation` member function causes the `TTimeScheduler` to delete the current `TOperation` object when it returns from the `Process` call that has been interrupted. Uses of the `DeleteInProcessOperation` are described in more detail below.

IsEmpty

The `IsEmpty` member function returns `true` if the scheduler is empty.

IsValid

The `IsValid` member function returns `false` if the scheduler was not constructed correctly. It should be called after creating the scheduler to verify that it constructed correctly. If it returns `false`, the scheduler should be deleted and not used.

RerunInProcessOperation

The `RerunInProcessOperation` member function causes `TTimeScheduler` to rerun the `Process` function for the current `TOperation` object when it returns from the `Process` call that has been interrupted. Uses of the `RerunInProcessOperation` are described in more detail below.

Reschedule

The `Reschedule` member function reschedules the operation on the `TTimeScheduler` object. Use `Reschedule` when an operation is already scheduled and you want to change the time that it will be fired. Using `Reschedule` avoids the necessity of going through the steps of first removing the operation, then setting its time, and then calling `Schedule`.

`Remove`

The `Remove` member function returns the specified operation from the scheduler.

`Schedule`

The `Schedule` member function schedules the specified operation. It will be run at the time specified in the operation.

`SetAutoReschedule`

The `SetAutoReschedule` member function sets the `TTimeScheduler` mode flag to enable or disable the `TTimeScheduler` class's auto-reschedule feature. Auto-rescheduling makes it possible for a `TOperation` object to reschedule itself automatically for another timer period simply by changing its own time field to a nonzero value.

Auto-rescheduling can be useful when an operation must perform multiple retries or simply wants to repeat the given operation. This strategy is sometimes convenient because `TOperation` objects that use the auto-reschedule feature do not have to know which `TTimeScheduler` they are on. A possible shortcoming of this strategy is that a `TOperation` object which is on an auto-reschedule `TTimeScheduler` can delete itself only under very controlled conditions.

The default state of the auto-reschedule feature is `false`, or disabled. When the auto-reschedule feature is disabled, a `TOperation` object is responsible for rescheduling itself.

See also `TScheduler`
`TOperation`

TTimeSchedulerExample on the *ASLM Examples* disk

TTimeStamp

The `TTimeStamp` class is used to get a time stamp of the CPU's current clock value.

The `TTimeStamp` class has the following inheritance:

```
TDynamic  -->  TMatchObject  -->  TDoubleLong  -->
                                   TTime  -->  TTimeStamp
```

Description

The time stamp value is set whenever the `TTimeStamp` object is constructed or whenever the `SetTimeStamp` member function is called. You can use `TTimeStamp` to compare the elapsed time between two time stamps by simply subtracting one from the other. The resolution of the time stamp is determined by the hardware. Some older machines only give a resolution of 1/60 second.

Since `TTimeStamp` is a `TTime` subclass, it can perform 64-bit time math to calculate the difference between two `TTimeStamps`.

Declarations

```
#define kTTimeStampID "slm:supp$tstm,1.1"


                    TTimeStamp();
virtual             ~ TTimeStamp();

virtual   void      SetTimeStamp();
```

Member functions

SetTimeStamp

The `SetTimeStamp` member function sets the time stamp value to the current time.

See also

`TTime`

TTimeExample on the *ASLM Examples* disk

TToken

The `TToken` class carries an object and gives it an ID.

The `TToken` class has the following inheritance:

```
TDynamic  -->  TMatchObject  -->  TToken
```

Description

Object arbitration is made possible by an object called a token, which maintains and provides information about objects. A token ID is an object made up of a type ID and an instance ID.

The type ID is determined by the developer of the shared library in which the token ID is used. To avoid name collisions, a token ID should uniquely identify the type of the token that it identifies in the context of the `TArbitrator` object for which the token registered. If more than one token of the same type is registered, an instance ID should be used to identify each instance. The type and instance IDs are separated by a dollar sign (`$`). Tokens registered with the global `TArbitrator` should have type IDs that are registered with Apple Computer's Developer Technical Support (DTS). It is sufficient to register a creator ID—for example, you might register the creator ID `'eesp'` for the an organization called the Excellent Enterprise Systems Protocols group.

Suppose, for example, that `eesp:sport$A` and `eesp:sport$B` are token IDs for the A and B serial ports on the Macintosh. In this case, `eesp:sport$` is the token for a serial port, and is the keyword that is used when the tokens are registered. Thus, a request for `eesp:sport$` is satisfied by either port. However, a request for `eesp:sport$A` is satisfied only by serial port A.

Normally, `TToken` objects are created automatically by calling `TArbitrator::RegisterObject`, but they can also be created by calling `TArbitrator:: NewToken`, in which case the token can only be used with the arbitrator that created it. Deleting a `TToken` will cause it to be automatically unregistered from the arbitrator that it is registered with. This is the way tokens are normally unregistered.

For more information on `TToken` and object arbitration, see "Object Arbitration Classes" in Chapter 8, "ASLM Utility Class Categories." The descriptions of the member functions below assume that you have already read this section and understand how object arbitration works. For details on the other object arbitration classes, see "TArbitrator," "TMethodNotifier," "TNotifier," "TProcNotifier," and "TRequestToken" in this chapter.

Continued on following page ▶

Declarations

```
#define kTTokenID              "!$tokn,1.1"

                               TToken();
                               TToken(const char*);
virtual                        ~ TToken();

// TMatchObject member functions

virtual   Boolean              IsEqual(const void*) const;
virtual   unsigned long        Hash() const;

// new member functions

virtual   Boolean              Get(TokenRequestType);
virtual   void                 Release();
virtual   Boolean              Request(TRequestToken*);
virtual   Boolean              Notify(TRequestToken*);
virtual   TokenRequestType     GetRequestType() const;

virtual   void                 SetID(const char*);
          const char*          GetID() const;

          void*                GetObject() const;
          void                 SetObject(void* theObject);

          TNotifier*           GetNotifier() const;
          void                 SetNotifier(TNotifier*);

          long                 GetUseCount() const;
```

Member functions

Get

The `Get` member function is used to claim the token. When you already have a pointer to the token, it can be used instead of requesting the token from its arbitrator. It can be useful if you have a token but do not have it claimed yet. You will want to do this when you are about to register a token using `TArbitrator::RegisterToken`, and you do not want to give up the token to anyone who already has an outstanding request for it. You might also do this if you do not have the token claimed, but you want to delete it if no one has it claimed already. In this case you will need to use a `TokenRequestType` of `kExclusiveTokenRequest`.

GetID
SetID

Each token has an ID associated with it called the object ID or token ID. The id string is allocated automatically when the token is created by calling `TArbitrator::RegisterObject` or `TArbitrator::NewToken`. If a token is not currently registered with an arbitrator, you can change its id by calling `SetID`. This will automatically delete the old id string and allocate a new one. Do not ever change the id string while the token is currently registered. You can also retrieve the id string at any time by calling `GetID`.

GetObject
SetObject

The `GetObject` and `SetObject` member functions get and set the object associated with the token. If the token is actually a `TRequestToken`, the object returned will be the requested token if the request has been satisfied. `NULL` is returned if the request has not been satisfied yet. There is normally no reason to call `SetObject`.

GetNotifier
SetNotifier

Tokens have a `TNotifier` object associated with them so that the token owner can be notified of certain events. If the token is actually a `TRequestToken`, the notifier is used to notify the requester that the requested token is available. If the token is a normal `TToken` registered with an arbitrator, the notifier is used to notify the token owner when an active request has been made for the token. Token owners normally setup a notifier for the token if they have exclusive access to the token and they are willing to give up the token if someone else requests it.

Token owners can set the token's notifier by calling `SetNotifier`. Before giving up a token, token owners should call `SetNotifier(NULL)` to remove the notifier.

Token requesters can set the notifier of their `TRequestToken` by calling `SetNotifier` or by passing the `TNotifier` object to `TArbitrator::PassiveRequest` or `TArbitrator::ActiveRequest.`

GetUseCount

`GetUseCount` returns the number of shared owners of the token. If the token is owned exclusively by someone, `-1` is returned.

Continued on following page ▶

`GetRequestType`

`GetRequestType` is used to determine if the `TToken` object is actually a `TRequestToken` object, and if it is a `TRequestToken` object, to determine what the request type of the token is. It is mainly used internally by `TToken`.

`Notify`

`Notify` is used to notify the exclusive owner of the token that there is a request for the token. The `TRequestToken` parameter passed in is the requester of the token. It returns `true` if the owner of the token had attached a notifier to the token and returns `false` otherwise. `TRequestToken::GetObject` can be used to determine if the owner gave up the token. `Notify` is mainly used internally by the object arbitration classes and does not have much use otherwise.

`Release`

`Release` is used to release a token. It releases the owner's claim on the token.

`Request`

`Request` is used to claim the token for the requester specified by the `TRequestToken` parameter. If it returns `true`, then either the token was successfully claimed or the token is already claimed and the owner had a notifier and was notified about the request. In this case, you should call the request token's `GetObject` method to see if the token was successfully claimed. This routine is mainly used internally by the object arbitration classes. It is only useful if you have both the token to be claimed and a request token that wants to claim the token.

See also

`TNotifier`
`TMethodNotifier`
`TProcNotifier`
`TRequestToken`
`TTokenNotification`

"Object arbitration classes" in Chapter 8, "ASLM Utility Class Categories."

TArbitratorExample1, TArbitratorExample2, and TArbitratorExample3 examples on the ASLM Examples disk

TTokenNotification

The TTokenNotification class is used with object arbitration to pass information to a client's notification function.

The TTokenNotification class has no parent class.

Description

The TTokenNotification class is a simple inline class that is used to return notification information to a client using object arbitration. It is passed in the notifyData parameter of the notifier's notify function.

When notifying a token owner that the token is being requested, GetToken is used to retrieve the requested token and GetRequestToken is used to retrieve the token that was used for the outstanding request. If you want to give up the token then call the request token's Give member function. You should not keep the request token unless you have an agreement with the client as part of your access protocol, and you must not keep the TTokenNotification.

When notifying the requester of a token that the token is available, GetRequestToken is used to retrieve the token that was used for the outstanding request. The token that was requested has already been claimed and is available by calling the Exchange or GetObject member functions of the request token.

For a full description of object arbitration and how TTokenNotification objects are used with object arbitration, see the "Object Arbitration Classes" section of Chapter 8, "ASLM Utility Class Categories."

Declarations

```
                TTokenNotification(TToken*, TRequestToken*);
                     ~TTokenNotification();

TToken*              GetToken();
TRequestToken*       GetRequestToken();
```

Member functions

GetToken

This function returns the token that was requested. This is true whether the owner of the token or the requester of the token is being notified.

GetRequestToken

This function returns the request token that is being used to handle the request.

Continued on following page ▶

See also `TNotifier`
`TMethodNotifier`
`TProcNotifier`

"Object arbitration classes" in Chapter 8, "ASLM Utility Class Categories."

TArbitratorExample1, TArbitratorExample2, and TArbitratorExample3 examples on the *ASLM Examples* disk.

TTraceLog

The TTraceLog abstract class can help you in debugging.

The TTraceLog class has the following inheritance:

```
TDynamic  -->  TTraceLog
```

Description

The TTraceLog class provides a Trace member function that has a parameter list equivalent to that of the C-language printf subroutine. When you send unformatted text to Trace, the text is formatted and is usually sent to a window, but this depends on the implementation of the TTraceLog subclass.

The GetGlobalTraceLog function provides a TTraceLog subclass that will send traces to the TraceMonitor’s Trace window. The SetGlobalTraceLog function can be used to change the global TTraceLog to your own TTraceLog subclass.

Declarations

```
#define kTTraceLogID "slm:dbug$tlog,1.1"


                    TTraceLog();
virtual             ~ TTraceLog();

virtual void        Trace(char *formatStr, ...) const;

     // New member functions

          Boolean   IsTraceLogOn() const;
          void      TraceLogOn();
          void      TraceLogOff();

virtual void   TraceFormatted(char* outstr) const = 0;
virtual void   TraceUnformatted(void* argp) const;
```

Member functions

IsTraceLogOn

The IsTraceLogOn member function returns true if tracing is turned on for the TTraceLog object.

Trace

The Trace member function calls TraceUnformatted and passes the address of the FormatStr parameter as the parameter for TraceUnformatted. The Trace member function, like printf, takes an unformatted string with multiple parameters.

Continued on following page ▶

TraceFormatted

The `TraceFormatted` member function displays the trace string passed to it. This is the only member function that a `TTraceLog` subclass must implement. The default global trace log sends the trace to the TraceMonitor's Trace window.

TraceUnformatted

The `TraceUnformatted` member function formats the trace to get the actual string to output. It then calls `TraceFormatted` to output the trace string.

TraceLogOn
TraceLogOff

You can turn a `TTraceLog` object's tracing on and off by calling the `TraceLogOn` and `TraceLogOff` member functions. You can also turn tracing on and off for any object that inherits from `TDynamic` by calling `TDynamic::TraceOn` and `TDynamic::TraceOff`, but only if the `TDynamic` subclass implements `TraceOn`, `TraceOff`, and `IsTraceOn`. By default, `TraceOn` and `TraceOff` do nothing, and `IsTraceOn` always returns `true`.

TUseCount

The `TUseCount` class is a data structure for maintaining a use-count value.

The `TUseCount` class has no parent class.

Description

The `TUseCount` class is defined inline and has only one field, the `fValue` field, so it requires no more overhead than any other use-count field unless a `short` was used instead of a `long`.

The `TUseCount` class returns `true` if `Increment` is called for the first time (that is, when the use count goes from 0 to 1) and when `Decrement` is called for the last time (when the use count goes from 1 to 0, or goes negative). Generally, when `Decrement` is called for the last time, that is a signal that some action must be taken. For example, if you want to delete an object after its use count goes to 0, a program can execute a statement such as:

```
if (myObject.fUseCount->Decrement()) delete myObject;
```

The advantage of using the `TUseCount` class is that the increment and decrement tests are atomic. If a program does not use the `TUseCount` class , but instead executes a statement such as:

```
if (--useCount <= 0)
```

The MPW compiler decrements the location of `useCount`, and then makes a separate test of the location. If the use count is changed at interrupt time after the first instruction is issued, but before the second, other routines may assume that they have incremented the use count from `0` to `1` or have decremented it from `1` to `0`, which can cause problems.

In order to work, `TUseCount` must store a value that is one less then the actual use count. Thus, when the use count is `0`, the value stored is actually `-1`. Therefore, `SetUseCount` is used to set the use count and `SetValue` is used to set the value (the use count `-1`).

Continued on following page ▶

Declarations

```
struct TUseCount
{
   void     SetValue(long);
   void     SetUseCount(long);
   long     GetValue() const;
   long     GetUseCount() const;
   void     Init();
   Boolean  Increment();  // Returns True if first time
   Boolean  Decrement();  // Returns True if back to unused
   Boolean  IsUnused() const;

   long     fValue;
};
```

Member functions

Increment
Decrement

The `TUseCount` member function returns `true` if `Increment` is called for the first time and when `Decrement` is called for the last time. When `Decrement` is called for the last time, it is a signal that some action must be taken.

Init

The `Init` member function is used to reset the use count to “unused.” In other words, the use count is set to 0 (value set to `-1`).

IsUnused

The `IsUnused` member function returns `true` if the use count is currently “unused” (the use count is `0`).

SetValue
GetValue

The `SetValue` and `GetValue` member functions are used to set and get the value (the use count `-1`).

SetUseCount
GetUseCount

The `SetUseCount` and `GetUseCount` member functions are used to set and get the use count.

IV Appendixes

Appendix A Header Files

To use the ASLM, a client must include certain ASLM header files. Five header files are provided with the ASLM:

- LibraryManager.h
- LibraryManagerClasses.h
- LibraryManagerUtilities.h
- GlobalNew.h
- TestTool.h

This appendix provides a general description of the contents of the ASLM header files.

LibraryManager.h

The LibraryManager.h header file contains essential interfaces for using the ASLM. Declarations in LibraryManager.h include the `TLibraryManager`, `TDynamic`, and `TClassID` classes; error codes; macros for exception handling; and some function declarations, including the declarations of `InitLibraryManager` and `NewObject`. The `TLibraryManager`, `TDynamic`, and `TClassID` classes are described in Chapter 9, “Utility Classes and Member Functions.” The C interface for `TLibraryManager` is described in Chapter 7 “ASLM Utilities.”

You can include the LibraryManager.h file in both C and C++ programs.

LibraryManagerClasses.h

The LibraryManagerClasses.h header file contains all the ASLM classes that are not defined in LibraryManager.h. If you write an application that uses or subclasses any classes declared in LibraryManagerClasses.h, the application must include the LibraryManagerClasses.h file. The LibraryManagerClasses.h file also contains function declarations that deal with certain classes defined in the file, such as `GetGlobalArbitrator` and `GetGlobalTaskScheduler`.

You can include the LibraryManagerClasses.h file in both C and C++ programs. However, most C programs will not need this file.

LibraryManagerUtilities.h

The LibraryManagerUtilities.h header file contains the interface to many of the utility functions and macros provided with the ASLM. The functions and declarations included in the LibraryManagerUtilities.h header file are described in Chapter 7, “ASLM Utilities”.

You can include the LibraryManagerUtilities.h file in both C and C++ programs.

GlobalNew.h

The GlobalNew.h header file contains the interface to the ASLM global `new` and `delete` operators, which allocate memory from pools rather than from the free store conventionally used in C++ programs. For more details on memory pools and the `new` and `delete` operators, see “Memory Management Classes” in Chapter 8 and “Using the ASLM Global `new` and `delete` Operators” in Chapter 6.

You can include the GlobalNew.h file only in C++ programs.

TestTool.h

The TestTool.h header file contains the interface to the `TTestTool` class, a class used by the MPW tool TestTool provided with the ASLM. `TTestTool` is a base class used for writing ASLM classes that can be used as test modules. TestTool creates and executes `TTestTool` subclasses. TestTool allows you to load and unload the ASLM, and load and run tests implemented by classes descended from `TTestTool`. You can also specify options to be passed on to the loaded objects.

TestTool is provided in executable form in the Tools folder on the *ASLM Developer Tools* disk, and is provided in source code form on the *ASLM Examples* disk. For instructions on building and using TestTool, see Appendix B, “ASLM Utility Programs.”

You can include the TestTool.h file only in C++ programs.

Appendix B ASLM Utility Programs

The ASLM includes several utility programs that demonstrate how shared libraries work and perform a variety of useful tasks. The source code for these programs has been provided in case you want to examine them or make use of them in your own clients and shared libraries.

The ASLM's utility programs are located on the *ASLM Debugging Tools* and the source code can be found on the *ASLM Examples* disks. This appendix describes the following utility programs and explains how to use them:

- LibraryManagerTest1
- LibraryManagerTest2
- Inspector
- TestTool
- TraceMonitor

For information on how to build the utility programs, refer to the section "Building the Examples" in Appendix C "Using the Example Programs."

LibraryManagerTest1 and LibraryManagerTest2

LibraryManagerTest1 and LibraryManagerTest2 are MPW tools that demonstrate how you can write shared libraries and clients in C++ and C. These tools can also perform a quick test of the ASLM so that you can tell whether the ASLM is working properly. Source code for the tools are provided in the ExampleLibrary folder on the *ASLM Examples* disk. The executable code is in the LibraryManagerTest folder on the *ASLM Dubugging Tools* disk.

Both LibraryManagerTest1 and LibraryManagerTest2 rely on a shared library named ExampleLibrary. The source code that builds ExampleLibrary is on the *ASLM Examples* disk, along with the source code for the two tools. A copy of the library that is already built is on the *ASLM Debugging Tools* disk, along with the executable LibraryManagerTest1 MPW tool.

To run the LibraryManagerTest1 and LibraryManagerTest2 tools, you must copy them into your MPW Tools folder. You must also copy the ExampleLibrary file into your system Extensions folder. This is described in “Installing the Debugging Tools” in Chapter 3, “ASLM Installation.”

The makefile in the ExampleLibrary folder builds two MPW tools: LibraryManagerTest1, which tests ExampleLibrary, and LibraryManagerTest2, a C version of LibraryManagerTest1. The same makefile builds a shared library named ExampleLibrary, which is used by the LibraryManagerTest1 and LibraryManagerTest2 tools.

The LibraryManagerTest1 and ExampleLibrary files supplied on the *ASLM Debugging Tools* disk are identical to the ones built using the makefile in the ExampleLibrary folder.

How LibraryManagerTest1 and LibraryManagerTest2 Work

Functionally, LibraryManagerTest1 and LibraryManagerTest2 are almost identical. LibraryManagerTest1 is written in C++, while LibraryMangerTest2 is written in C and shows how to call methods of classes from C.

Both tools instantiate objects in the ExampleLibrary and call functions that are implemented in a function set in the ExampleLibrary file.

The syntax of the LibraryManagerTest1 and LibraryManagerTest2 commands is:

```
LibraryManagerTest1 [-v] [-t 0|1] [-c nReps] [-s] [-l] [-x]
                    -i classID
LibraryManagerTest2 [-v] [-t 0|1] [-c nReps] [-s] [-l] [-x]
                    -i classID
```

where:

`-v`	Turns on verbose mode, which prints progress messages in the MPW worksheet. Default is off.
`-t`	Turns on tracing. Default is off.
`-c nReps`	Sets the number of times a test loop will run. The nReps variable is a positive integer that can be set to the number of times the test will iterate its while loop.
`-s`	Unloads the ASLM.
`-l`	Loads the ASLM.
`-x`	Turns on debugging.
`-i classID`	Tests GetClassInfo with the specified class ID.

Running LibraryManagerTest1 or LibraryManagerTest2

You can run the LibraryManagerTest1 file located in either the *ASLM Debugging Tools* disk or the *ASLM Examples* disk. Before running LibraryManagerTest1 or LibraryManagerTest2 do the following.

- Drag LibraryManagerTest1 or LibraryManagerTest2 into your MPW Tools folder.
- Drag the ExampleLibrary from the Built folder file into your Extensions folder.

To run the LibraryManagerTest1 tool, execute the LibraryManagerTest1 command by entering this command:

```
LibraryManagerTest1
```

If the ASLM is installed in your system and is operating properly, LibraryManagerTest1 prints an analysis similar to the following:

```
Hello(ulong&):Hello
startticks = 283301
Hello(ulong*):Hello
startticks = 283309
HelloC(ulong*):Hello
startticks = 283316
HelloPascal(ulong&):Hello
startticks = 283323
100000 Iterations of Hello: 180
100000 Iterations of local Hello: 144
Elapsed ticks (according to 'C' interface): 151
Elapsed ticks: (according to Pascal interface): 155
```

The LibraryManagerTest2 tool also tests the ASLM and provides output similar to that of the LibraryManagerTest1 tool shown earlier. To run the LibraryManagerTest2 tool enter the command:

```
LibraryManagerTest2
```

The Inspector application

The Inspector is an application that helps you debug shared libraries. The Inspector lets you inspect objects that are implemented in shared libraries and is a good example of how to write a shared library that displays windows, menus, and dialog boxes. The source code is in the Inspector folder on the *ASLM Examples* disk. The Inspector also allows you to load or unload the ASLM, register a shared library file, or register a shared library file folder.

The makefile in the Inspector folder builds the Inspector application along with InspectorLibrary and WindowStackerLibrary (two shared libraries required by the Inspector application).

In case you do not choose to build the Inspector example, an executable copy of the Inspector application and its libraries—InspectorLibrary and WindowStackerLibrary— can be found, already built, on the *ASLM Debugging Tools* disk.

For information on how to register C++ objects with the Inspector, see “Registering C++ Objects with the Inspector” in Chapter 7, “ASLM Utilities.”

Running the Inspector

Before running the Inspector, you must drag the InspectorLibrary and WindowStackerLibrary files into your system Extensions file.

To run the Inspector:

1 Run the Inspector by double-clicking the application icon in the Finder.

2 If the ASLM is loaded, the Inspector starts and displays three or more windows.

How the Inspector works

Each window that the Inspector displays represents a C++ class. The title of each window is the class ID for the class that the window represents.

In the content region of each window, there is a list of instantiated objects that belong to the class represented by the window. These objects are registered by calling `RegisterDynamicObject`.

Next to the name of each object, there is a text string. These strings are returned by each object's `GetVerboseName` method. The `GetVerboseName` method is a `TDynamic` class method that can be overridden by its subclasses. (For more information about the `TDynamic` class and its methods, see Chapter 9, "Utility Classes and Member Functions.")

The Inspector always displays at least three windows, each of which represents a class used internally by the ASLM. The three main windows that the Inspector displays can be useful when you want to see which shared libraries, shared library files, and classes are currently recognized by the ASLM. The windows also provide some useful information about each class:

- The !$file window contains one object for each shared library file. The information supplied for the object includes the object's filename, directory ID, and volume refNum.
- The !$libr window contains one object for each shared library. The information in this window includes the *use count* for the library (a use count of 0 means the library is not in use and is not loaded), the library's unformatted version number, and information about the library's library file.
- The !$clss window contains one object for each class in a shared library. The information presented in this window includes the class' flags, use count, and class ID. Function sets are also displayed in this window.

  The use count in the !$clss window is not necessarily the number of instances of the class. Although the use count is incremented each time an instance is created, it is also incremented each time a library containing a subclass of the class is loaded, even if the subclass is not instantiated.

Here are the flags you see in the Inspector. Flags are in hex. The values in the inspector are also in hex and represent the sum of all the flags that are set.

The following are the flags for the !$clss window:

1:	the class has the `preload` flag set
2:	the class has the `NewObject` flag set
4:	the class is actually a function set
8:	the class has a virtual destructor
10:	the class is a *dummy* function set resulting from the use of the `interfaceID=` option for function sets
40:	the class uses multiple inheritance
80:	the class is an ASLM root class

The following are the flags for the !$libr window:

1:	the library is built with `flags=preload`
2:	the library is built with `memory=client` rather than `memory=local`.
4:	the library is built with `flags=noSegUnload` rather than `flags=segUnload`
8:	the library is built with `flags=loaddeps` or `flags=forcedeps`
10:	the library is built with `flags=forcedeps`
20:	the library is built with `flags=loadself`
80:	the library uses per client data
100:	the library is built with `heap=temp`
200:	the library is built with `heap=system`
400:	the library is built with `heap=HOLD`
100 and 200:	the library is built with `heap=application`

Inspector menus

The Inspector has the standard Macintosh File and Edit menus (although the Edit menu is not activated), a Windows menu, and a Commands menu. From the Windows menu, you can select and stack windows. The Tile Windows command under the Windows menu is currently not implemented.

The Commands menu contains commands to reload and unload the ASLM, to turn tracing on and off, and to switch from the Inspector's normal mode to a bare-bones simple program mode that has only a File menu and an Edit menu and that does not display windows. When in this mode, the Inspector does not require its shared libraries and does not require that the ASLM be loaded.

The Unload Library Manager command and the Reload Library Manager commands are used mainly for testing purposes.

WARNING Choosing Unload Library Manager during normal use of the ASLM can cause any client currently using the ASLM to crash.

The Goto Simple Program command in the Commands menu places the Inspector in its simple program mode, which does not display windows and does not require the ASLM to be loaded. This mode lets you launch the Inspector without having the ASLM present. You can then load the ASLM from within the Inspector.

Choosing the Goto Simple Program command is not the only way to put the Inspector into simple program mode. If the ASLM is not loaded when you launch the Inspector application, the Inspector goes into simple program mode automatically. Another way to put the Inspector into simple program mode is to unload the ASLM while the Inspector is running. You can do that by choosing the Unload Library Manager command.

When the Inspector is in simple program mode, only the File menu is active. If the ASLM is loaded, you can take the Inspector out of Simple Program mode by choosing Goto Real Program (the mode with windows), or you can unload the ASLM by choosing Unload Library Manager (see the warning above). If the ASLM is not loaded, you can load it, but only if it was loaded at boot time and has since been unloaded.

The Register Folder menu item will register the folder you select as a registered library file folder by using the `RegisterLibraryFileFolder` function. You can unregister the folder by using Unregister Folder. Likewise, Register File registers the library file you select with the ASLM by using the `RegisterLibraryFile` function, and Unregister File allows you to unregister the library file.

TestTool

TestTool is an MPW tool that allows you to load and unload the ASLM and test classes included with the ASLM and shared libraries that you develop yourself.

The makefile in the TestTools folder builds TestTool and a shared library named TestLibrary. The source code files that are used to build TestTool and TestLibrary are on the *ASLM Examples* disk.

The TestTool file and the TestLibrary file that are built using the makefile on the *ASLM Examples* disk are identical to the executable copies of TestTool and TestLibrary that are supplied in the TestTool folder on the *ASLM Debugging Tools* Disk.

Using TestTool

Before running TestTool, drag the TestTool file into your MPW Tools folder, and the TestLibrary file into your system Extensions folder.

You can run TestTool by executing the TestTool command. To execute the TestTool command, pass the name of the class to do the testing as a parameter on the command line. TestTool then runs the tests by calling the object's methods. The class must inherit from the `TTestTool` class, which is declared in the TestTool.h header file. You can write your own TTestTool subclasses if you like.

The syntax of the TestTool command is:

```
TestTool [v] [-t0|1] [-n nReps] [-s] [-l] [-x]  [-p]
          [-c ClassID] [-a]
```

where:

`-v`	Turns on verbose mode, which prints progress messages in the MPW worksheet. Default is off.
`-t`	Turns on tracing Default is off.
`-n nReps`	Sets the number of times a test loop will run. The `nReps` variable is a positive integer that can be set to the number of times the test will iterate its `while` loop.
`-s`	Unloads the ASLM.
`-l`	Loads the ASLM.
`-x`	Turns on debugging.
`-p`	Does not allow the memory pool to grow.

`-c classID`	Runs tests using the specified class.
`-o`	Remaining arguments are passed to `<YourTestTool>::InitTest`
`-a`	Runs all tests.

This is an example of a TestTool command:

```
TestTool -v -t -n 5 -c TTestTaskScheduler
```

TestTool classes

The class ID variable that you specify in the `-c` option should not contain a prefix because all `TTestTool` subclasses use a class ID with the `slm:test$` prefix. Note that the class ID variable is case sensitive. (For more information on class IDs, see "TClassID" in Chapter 9, "Utility Classes and Member Functions.")

The TestLibrary contains the following classes that you can use with TestTool:

TTestAbitrator	TTestNoVTable
TTestTaskScheduler	TTestRandom
TTestTimeScheduler	TTestStandardPool
TTestTimeStamp	TTestTimings
TTestExceptions	TTestPriorityList
TTestFSet	TTestAllocLinkedList
TTestHashList	TTestLinkedList
TTestMisc	

The TraceMonitor application

The TraceMonitor application displays traces that are sent by ASLM clients using the `Trace` routine. The `Trace` routine sends output to the currently installed global trace log. The global trace log that the ASLM installs, sends the traces to the TraceMonitor application which displays them in its main window. Traces can be useful for debugging shared libraries since they do not have access to any other type of debugging window. For more information, see "Using the Global TraceLog" in Chapter 7, "ASLM Utilities."

Appendix C Using the Example Programs

The ASLM package contains a collection of example programs that can help you create and build clients and shared libraries. Source code is provided so that you can examine, and then compile and link into executable clients and shared libraries. Some of the code samples in this document are taken from these examples.

The examples are in seven folders on the *ASLM Examples* Disk:

- Example Tools
- ExampleLibrary
- FunctionSetInfo
- Inspector
- Sample INIT
- Sample Apps
- TestTools

The examples in the folders ExampleLibrary, Inspector, and TestTools are utilities described in Appendix B "ASLM Utility Programs."

The programs in the folders Sample Apps, Sample INIT, ExampleTools, and FunctionSetInfo are examples of clients and shared libraries written in C++, C, and Pascal, and a sample Extension (INIT) that makes use of the ASLM.

Each of the example folders contains a makefile and a set of three folders named Sources (containing the source files), Objects (containing the object files), and Built (containing the built files). The Example Tools folder has a `BuildExample` script instead of a makefile. If an example has more than one makefile, execute the makefile that builds the shared libraries first and the tools or applications second.

The Sample Apps folder

The sample programs in the Sample Apps folder are patterned after the Sample.c and Sample.p programs that are supplied with MPW. Like Sample.c and Sample.p, each sample program in the Sample Apps folder displays a single window on the screen. The window contains a picture of a traffic light. By either clicking inside the picture or selecting a menu item, you can make the traffic light switch back and forth between red and green (or between two different patterns if you do not have a color Macintosh).

Each version of the program provided with the ASLM is divided into two parts: a client and a shared library. Code that is not likely to be useful in other programs (in the opinion of its author) was placed in the client section of each program. Code that was believed more likely to find its way into other programs was placed the shared library associated with each client.

The sample programs in the Sample Apps folder are

- CSample, a client and shared library written in C
- CPlusSample, a client and shared library written in C++
- CCPlusSample, a client written in C and a shared library written in C++
- PSample, a client and shared library written in Pascal

Before running any of the sample applications, the shared library file that is built with the application must be placed in the Extensions folder.

The Sample INIT folder

The Sample INIT folder contains an example of an INIT that uses a shared library. The example includes a shared library that implements the `ShowINIT` function commonly used by INITs. The INIT in this example calls the `ShowINIT` function in the shared library.

The source files in the Sample INIT folder include

- SampleINIT.c, a C-language example that shows how Extensions (INITs) can use the ASLM
- SampleINIT.r
- ShowINITLibrary.c
- ShowINITLibrary.exp
- ShowINITLibrary.h
- ShowINITLibrary.r

Before rebooting your machine, the SampleINIT and ShowINITLibrary files must be placed in your Extensions folder.

The FunctionSetInfo folder

The FunctionSetInfo folder contains an example of how to find all function sets that have a common interface ID by using the `GetFunctionSetInfo` function. In the example, there are two function sets, `MathFSetAdd` and `MathFSetSub`, which share a common interface. The `TestMathSet` MPW tool uses `GetFunctionSetInfo` to find these function sets and then calls the `MathFunction1` and `MathFunction2` functions in each function set by using `GetFunctionPointer`. After building the example, the MathLibrary file must be placed in the Extensions folder before running the TestMathFSet tool.

The source files in the FunctionSetInfo folder include

- TestMathSet.c, which is an MPW tool that demonstrates how to use `GetFunctionSetInfo`
- MathFSetAdd.c, which is the implementation of the `MathFSetAdd` function set
- MathFSetSub.c, which is the implementation of the `MathFSetSub` function set

The Example Tools folder

The Example Tools folder contains a large assortment of tools that demonstrate how you can use the utility classes supplied with the ASLM in your clients and shared libraries.

The example tools that are built have file names that are, for the most part, self-explanatory. The programs are

LibraryManagerExample	TPoolNotifierExample
TArbitratorExample1	TPriorityListExample
TArbitratorExample2	TPrioritySchedulerExample
TArbitratorExample3	TProcNotifierExample
TArrayExample	TSerialSchedulerExample
TClassInfoExample	TSimpleListExample
TInterruptSchedulerExample	TTaskSchedulerExample
TLinkedListExample	TTimeExample
TMacSemaphoreExample	TTimeSchedulerExample
TMethodNotifierExample	TTokenExample

Building the examples

When you build an example you must always copy its shared library (if it has one) into the Extensions folder before running the example (unless you want to see an example of ASLM exception handling).

To build the example programs, do the following:

1. Set {SLMInterfaces} to the directory where the ASLM interface files are located, and export SLMInterfaces.
2. Set {SLMLibraries} to the folder where the ASLM MPW libraries are located, and export SLMLibraries.
3. Either add the directory where the ASLM tools are located to your {Commands} path or copy the tools into the MPW Scripts and Tools folders.

The following sample code builds the example programs. This sample assumes that the *ASLM Developer Tools* disk is located on a hard drive named HD:

```
set SLMInterfaces "HD:ASLM1.1: Developer Tools:Interfaces:"
export SLMInterfaces
set SLMLibraries "HD:ASLM1.1: Developer Tools:Libraries:"
export SLMLibraries
set Commands "HD:ASLM1.1: Developer Tools:Tools:,{Commands}"
```

You must be in the directory containing the makefile for the example to build. If you want to build all the examples at once, set the current directory to the ASLM Examples folder and then execute the following:

```
directory :ExampleLibrary
make > make.out
make.out

directory ::Inspector
make -f makefile.libs > make.out
make.out
make > make.out
make.out

directory ::TestTools
make  > make.out
make.out
make  -f makefile.tools > make.out
make.out
```

```
directory '::Sample INIT:'
make > make.out
make.out

directory '::FunctionSetInfo:'
make > make.out
make.out

directory '::Example Tools:'
BuildExample -a

directory '::Sample Apps:'
directory ':CSample:'
make > make.out
make.out
directory '::PSample:'
make > make.out
make.out
directory '::CCPlusSample:'
make > make.out
make.out
directory '::CPlusSample:'
make > make.out
make.out
```

Building .SYM files for clients, libraries, and tools

To build the example programs with .SYM files, you must execute the following command before running the makefiles for the examples.

```
Set SymbolOption "-sym on"; export SymbolOption
```

The .SYM files that you create in this manner are placed in the Built folder of the example that you are building. This will only work for the Inspector, ExampleLibrary, and TestTools examples.

Appendix D Versioning

When you write a shared library or a client, you can specify the version numbers of function sets and classes implemented in the shared library or used by the client. You can place both the current version and the minimum supported version of a function set or class in the exports file of the library that you are writing. The class version information that you place in an exports file is represented by a range of numbers. For example, 1.0...1.2, indicates the oldest version (1.0) and the most recent version (1.2) supported by the class. Version numbers are referred to in terms of major, minor, and bug-fix. A period separates the major, minor, and bug-fix numbers. In version 3.5.2, the major number is 3, the minor number is 5, and the bug-fix number is 2.

When you create an object or call a function that is implemented in a shared library, the ASLM uses the class or function set with the newest version number that is also compatible with the version specified in the client object file with which the client linked.

When a client uses the `NewObject` function to create an instance of a class, the ASLM uses the class with the newest version number that is also compatible with the version specified in the class ID passed to `NewObject`. If the class ID contains no version number then the latest version is always used. The same is true when you use the `GetFunctionPointer` routine to get a pointer to a function in a function set. The function set ID determines which version of the function set will be used.

You can place version information in function set IDs and class IDs. In fact, both function set IDs and class IDs should contain version numbers. For more information on this topic, see “TClassID” in Chapter 9, “Utility Classes and Member Functions.”

Shared libraries also have version numbers. When you are developing a shared library, you can assign version numbers to progressive versions of a shared library. The version number should also be part of the library ID so that each version of the shared library will have a unique library ID. The library's version number also serves as the default version for function sets and classes that do not specify a version. Therefore, when you assign a version number to a shared library, each function set and class in the library that does not have its own version number is assigned the version number of its shared library.

When there are multiple shared libraries with the same library ID, the ASLM registers all of the shared libraries, however, only the function sets and classes in one of the libraries will be used. The others are marked as duplicates and are not used, even if different version numbers are used for the libraries, function sets, or classes.

How versioning works

When a shared library is built, information about the version of each function set and class is placed in the library's client object (.cl.o) file. Therefore, the client object file with which a client links determines which version of the function set or class is used by the client. For example, assume that a .cl.o file contains a function set or class that is designated as version 1.2. When the class is created (or when a function in the function set is called) the ASLM looks for the function set or class with the highest version number that supports the version in the .cl.o file that the client linked with—in this case, version 1.2.

Now assume that three versions of a class exist: version 1.2 (which supports versions 1.0…1.2), version 1.4 (which supports versions 1.1…1.4), and version 1.5 (which supports versions 1.3…1.5). If a client links with the client object file for the 1.2 version of the function set or class, the ASLM chooses version 1.4 because it is newer than version 1.2, and because version 1.5 does not support version 1.2. If the client linked with the client object file for the 1.0 version of the function set or class, version 1.2 of the function set or class is used because it is the only one that supports version 1.0.

It is possible for multiple versions of a function set or class to be in use at the same time. This can happen if the version ranges of the classes do not completely overlap (for example, if the available version ranges are 1.1…1.4 and 1.3…1.5) or if an older version of a function set or class is in use when a new version is made available. The older version of the function set or class continues to be used by its existing clients but the newer version is used by any new clients that start up after the new function set or class is added.

Note: If a class does not have a virtual destructor, only one version of the class can be used at a time. This restriction is needed to ensure that the proper destructor is called when instances of the class are deleted.

Function sets can maintain backwards compatibility by always listing the function to be exported in the same order in the exports file and not changing the interfaces of existing functions. If this is done, new functions may be added to the function set and the version range of the function set can continue to include older version numbers.

Version numbers and subclasses in C++

If new data members or virtual member functions are added to a class, the user of the class is unaffected because the v-table offsets of the functions and the locations of the data members of the class that the user knows about remain the same.

However, subclassing a class that has added virtual member functions or data members has a definite impact on the subclass. If virtual member functions are added to a base class, they will be overwritten by the new virtual functions of the subclass. This is not necessarily a problem, provided the parent class does not call any of the new functions itself. If you pass an object that is an instance of the subclass to a function holding the new definition of the base class, the function may attempt to call new member functions of the class. Since this will probably fail, the function will then call the overwritten function in the subclass. Similar problems may arise when data members are added to a class.

To prevent this problem, a change in a class's major version number indicates that the class is no longer subclass compatible with previous versions of the class. Clients linked with the older version's client object file may instantiate the class and get the new class if the new class is backwards compatible. However, 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.

If the base class has its version numbers set correctly, subclass compatibility is all handled automatically. For example, assume you implemented version 1.0 of a class called `TFoo` that is subclassed by a class called `TSubFoo` (which is in another shared library), and then a newer version of `TFoo` is introduced that has added some virtual functions, but is otherwise compatible with version 1.0. As a result, the major version of `TFoo` must change, and you must use version 1.0…2.0 (not 1.0…1.1). When someone creates an instance of `TSubFoo`, `TSubFoo` will automatically use version 1.0 of `TFoo` since it was linked with version 1.0 of the client object file containing `TFoo` and it knows that version 2.0 is not subclass compatible.

Using the example above, if a client was linked with version 1.0 of the client object file containing `TFoo`, then it would automatically use version 2.0 of `TFoo` when it creates instances of `TFoo` using the default `new` operator. However, if the client created an instance of `TFoo` using a nondefault `new` operator (such as one where you explicitly specify the pool out of which to allocate memory) or if the object is a stack object then version 1.0 of `TFoo` is used automatically.

The reason version 1.0 is used in this instance is because the memory for the object is allocated by the client, and not by the constructor of the object as it is when using the default `new` operator. Since the client will not have any idea that the size of the object has grown, it needs to play it safe and only use a version that it knows is subclass compatible.

If there is a version incompatibility (that is, if there is no shared class with a compatible version), an exception is raised using the error code `kVersionError` or the error code `kNotFound`.

WARNING The `sizeof` function always returns the size of the class declared in the interface files with which a library or a client is compiled. However, if you use the `new` operator to create an instance of the class, you may get an object back that is bigger then the result of the `sizeof` function. This is possible if a newer version of the class exists, the newer version adds new data members, and indicates how to be compatible with the class you requested. This will not occur if the object created is a stack object or an embedded object. In this case, you get an object of the correct size. To ensure that you get the correct size of a class, call `GetObjectsSize` after creating an instance of the class.

The following table summarizes how to handle your class version numbers when you make changes to the class:

Change made to the class	Action needed for versioning
Virtual functions deleted	Change the class ID of the class
Data members deleted	Change the class ID of the class
Virtual functions added after the last virtual function	Increase the major version of the class
Data members added after the last data member	Increase the major version of the class
Implementation changed	Increase the minor or bug-fix version of the class
Non-virtual methods added	Increase the minor or bug-fix version of the class
A new constructor added after the last constructor	Increase the minor or bug-fix version of the class
Change made to a data member	If the data member is the same size, and you have no inline functions to it, just increase the minor or bug-fix version of the class. Otherwise, you need to change the class ID of the class.