SOMobjects™ for Mac™ OS

SOMobjects and System Object Model are trademarks of IBM Corporation.

Introduction

This document describes version 2.0.7 of SOMobjects™ for Mac OS which is based on the System Object Model version 2.0 from IBM.

The document assumes that the reader is familiar with the System Object Model as documented in IBM's SOMobjects Developer Toolkit: Users Guide, chapters 1–5 which for brevity will be called "IBM Users Guide" in the remainder of this Release Note. The IBM Users Guide, and other IBM documents can be found in the Documentation folder.

SOMobjects is an object-oriented programming technology for building, packaging, and manipulating binary class libraries. Traditionally, it has been difficult to produce shared libraries with an object-oriented C++ interface that provide binary compatibility from release to release. With SOMobjects, these limitations of the C++ runtime are removed. It is expected that SOMobjects will be used when an object-oriented API to a shared library is desired.

Interfaces to a class are specified in the CORBA[1] Interface Definition Language (IDL) which is described in the IBM Users Guide. Classes may be implemented in any language for which the developer has an IDL compiler. The client of a class may use a different language than the implementation. This release of SOMobjects™ for Mac OS includes support for development in C and C++.

Components

This release contains the following components:

- The runtime kernel, "SOMobjects™ for Mac OS", located in the "System Additions" folder.
- Kernel interfaces for C/ C++ and IDL, located in the {CIncludes} and {SOMIncludes} folders.
- The runtime kernel stub library, "somlib", located in the {SharedLibraries} folder.
- The IDL Compiler, `somc`, with emitters for C and C++, located in the {Scripts} folder with emitters in the {Tools} folder of the MPW environment.
- The PDL facility, `sompdl`, located in the {Tools} folder.

[1]CORBA, Common Object Request Broker Architecture is a standard of the Object Management Group (OMG).

- The MPW startup script, “UserStartup•somc”, located in the “MPW Additions” folder.
- Programming examples in the SOMExamples folder.

The SOMobjects™ for Mac OS Runtime Kernel

The runtime kernel for Power Macintosh and CFM-68K is a fat shared library consisting of the classes `SOMObject`, `SOMClass`, `SOMClassMgr` and `SOMRegisteredClassList`.

Interfaces

Interfaces to the kernel are presented as C/C++ and IDL headers. They are described in the reference section of this document.

Development Tools

SOMobjects IDL Compiler

The IDL compiler, `somc`, is an MPW script/tool which translates IDL files into C and C++ headers and skeleton source files. The compiler and IDL language are documented in Chapter 3 of the IBM Users Guide.

Syntax and Options

```
somc [ options ] source.idl…
```

Notes:

- The name used by IBM, sc is not available because it is already in use for the Symantec C compiler.
- Unlike the syntax of IBM’s sc, options may appear anywhere on the command line (before or after source files).
- Unlike the syntax of IBM’s sc, options under MPW are not case sensitive.

Options

-D define	Defines macro for `#ifdef` processing.
-D define=*value*	Defines macro with a value.
-e *emitterName…*	Specifies emitter(s) to run.

For example,

```
-e xh,xih,xc
```

runs the emitters xh, xih, and xc in the order given.

See page 6 of this document for a description of the emitters available with this release.

-keeptm Keeps temporary files (.ctm when using the C emitter, c, and .xtm when using the C++ emitter, xc).

-I *dirName* Specifies path to be searched first for include processing.

-m chkexcept Specifies the inclusion of exception result checking during generation of C++ .xh class bindings. This mechanism can be used to introduce an automatic "throw" or error handler of some sort, removing the need to manually check exception results after each method call.

-m corbastring Instructs the IDL compiler to translate the IDL data type `string` into C/C++ as `corbastring`, instead of passing it through into C/C++ unchanged (as `string`). This is to avoid a potential conflict should `string` become a reserved word.

-m cplusc Specifies the use of .C for C++ file suffixes (the default C++ file suffix is .cp).

-m cpluscpp Specifies the use of .cpp for C++ file suffixes (the default suffix for C++ files is .cp).

-m level2 Instructs the somidl compiler to allow method name ambiguities that arise due to multiple inheritance. Note that use of an ambiguous method name will require qualification.

-m *modifier* Passes a global modifier to compiler. All of the modifiers described in the IBM Users Guide, pages 40-48, 4-49 are supported except for `notc`, `pp=preprocessor`, and `nopp`.

-m nomethodids Specifies that the programmer does not want global Ids to be generated into the client bindings file. In some build environments this reduces client and implementation code size. It also makes the header file more readable and speeds compilation. Ids for method names are rarely used by programmers; if they are desired, they can be created individually instead of relying on the compiler to generate them. This option should generally be on, and is specified in the default UserStartup•somc.

-m noshortparents	Suppresses generation of SOMobjects 1.0 parent call through macro forms into the ih and xih (C and C++ implementation) header files. When Multiple Inheritance was added to SOMobjects, new parent call through macro forms were created. This optimization makes the header file more readable and speeds compilation. This option should generally be used, and is in the default UserStartup•somc.
-m notexported	This modifier specifies that the interface(s) being compiled are for internal use within one fragment only, that is, not to be automatically exported from the fragment that is being built, (and not to be imported from another fragment).
-m novastatics	Suppresses generation of static procedures for translating variable parameter list calls to methods that take a `va_list` as an argument in the .h (C client) header files. This can reduce client code size on some build systems.
-m useinheritedmacros	Specifies that the xih emitter (for C++ implementations) should not reintroduce ancestor methods at each class level. This makes the header file more readable and speeds compilation. This option should generally be used, and is in the default UserStartup•somc.
-o *outdir*	Specifies the output directory for emitted files. (Note that this does not specify the output file itself.)
-other *option*	Specifies other miscellaneous options for the IDL compiler. For details on miscellaneous options, see SOMobjects Developer Toolkit: Users Guide, section 4.6.
-p	Causes the IDL Compiler to produce private versions of its output files. This is necessary when producing implementation files, and unwanted when producing client files.
-r	Checks that names specified in the release order statement are valid method names.
-v	Uses verbose mode.
-w	Turns warnings off.

IDL Compiler Emitters

The following table describes the IDL compiler emitters supported. Unless otherwise stated, the Target File is created (i.e., overwritten).

Emitter Name	Target File	Purpose
xc	xxxx.cp (updated)	Update C++ main implementation file. Note: the C++ file extension can be changed via `-m cplusc` or `-m cpluscpp`
xtm	xxx.xtm	Create C++ main implementation skeleton file.
xih	xxx.xih	Create C++ main implementation header file.
xh	xxx.xh	Create C++ client header file.
c	xxx.c (updated)	Update C main implementation file.
ctm	xxx.ctm	Create C main implementation skeleton file.
ih	xxx.ih	Create C main implementation header file.
h	xxx.h	Create C client header file.
exp	xxx.exp	Create implementation exports file, which is supplied to the linker (required for PowerPC development, optional for CFM-68K)

Note: To generate a public IDL file, the `sompdl` tool is used rather than the somc compiler. (The pdl emitter is supplied, but considered obsolete.)

---

sompdl

This tool takes as input an interface definition file (.idl) and produces a similar file from which the declarations of private class members have been deleted. This tool should be used when developing shared libraries to produce public versions of .idl files for distribution.

Syntax

```
sompdl [options] file…
```

Options

-d *dir* Specifies a directory in which the object files are to be placed. The

output files are given the same name as the input files. If this option is omitted, the output files are named <*fileStem*>.pdl where *fileStem* is the file stem of the input file and are placed in the current working directory.

-f Specifies that output files are to replace existing files having the same name even if the existing files are read- only.

IDL Compiler Shell Variables

The somc compiler uses three Shell variables to specify default values :

- `SOMIncludes` search paths for .idl include files
- `SOMDefines` pre-defines for #ifdef processing
- `SOMOptions` -m modifiers for .idl file compilation

These are preset by UserStartup•somc as follows:

```
Set SOMIncludes "{MPW}Interfaces:SOMIncludes:"
Set SOMDefines "-D __SOMIDL__ -D __MAC__"
Set SOMOptions "-m addstar -m noint -m useinheritedmacros ∂
                -m nomethodids -m corbastring -m noshortparents"
```

If these options are changed, the standard pre-compiled header files will require regeneration. This is done by generating new .h and .xh headers for all the standard .idl files (in {SOMIncludes}). The targets go back into {CIncludes}. The following commands recompile the .idl files and places the resulting C and C++ headers into the {CIncludes} folder:

```
somc "{SOMIncludes}"≈.idl -e h -o "{CIncludes}"
somc "{SOMIncludes}"≈.idl -e xh -o "{CIncludes}"
```

Using SOMobjects™ for Mac OS

The runtime kernel provides services in the form of functions, macros and classes. There are two significant base classes: SOMObject from which all classes descend, and SOMClass, the base meta-class. The following section assumes an understanding of the relationship between these classes as documented in the IBM Users Guide.

Programmers access the kernel through its Application Programming Interface (API) and through compiler produced Bindings. Most of the required access is provided by the Bindings which are designed to interface efficiently with the kernel.

Allocating Objects

The simplest way to allocate objects is to use the C or C++ bindings generated by the IDL compiler; `New<className>` or `new <className>`.

The 2.0.7 API provides four principal functions for allocating objects directly: `somNewObject`, `somNewVersionedObject`, `somNewObjectByName` and `somNewObjectById`. The first two return a Static Reference while the last two return a Dynamic Reference. More information is available in the sections on Static and Dynamic Class Status.

Freeing objects

The recommended way to free an object reference is to use the kernel operation `somReleaseObjectReference`. The 2.0.7 SOMObject class supports both `somFree` and a new method, `somRelease`, however direct invocation of these methods can introduce cross-platform portability problems.

When the last reference to an object is released, that object will release its reference to its class implementation which can potentially cause the implementation code fragment to be unloaded. Care should be taken to avoid having objects free themselves because this can leave stack frames pointing to code which no longer exists.

Object References

The 2.0.7 APIs are reference based. It is particularly important to understand object ownership since a class' implementation may be dynamically unloaded any time it is considered not referenced. There are some simple guidelines you can follow which will make your programs more robust, thread safe, CORBA compliant and self-contained.

- If you obtain an object reference as a function or method return result, you are responsible for releasing it when no longer needed.
- If you are returning an object reference to your caller, it should be a new or duplicated reference which becomes the property of the caller.
- If you receive an object reference as a parameter which you'd like to hold on to, you must obtain a duplicate reference to it via the `somDuplicateReference` method.

In order to compare object references, do not perform pointer comparison. Use the `somCompareReference` method instead.

Class Objects

SOMobjects™ for Mac OS provides extended facilities for class object usage for clients with a specific need or desire to program with them. For clients who only access class objects for instance allocation, the new kernel APIs make their access unnecessary.

Access to class objects is obtained by the kernel operations `somNewClassReference`, `somNewVersionedClassReference` and `somGetDynamicClassReference`. Starting with an existing object, you can ask it for its class object by sending it a `somGetClass` message. The kernel SOMClassMgr object also provides access to class objects and is documented below. Each of these operations returns a class reference which must later be released via `somReleaseClassReference`.

In the System Object Model, classes are represented by class objects. Class objects are used to control certain aspects of class behavior, such as instance object allocation. Class objects can be obtained statically or dynamically. When a class is first loaded from the disk, it is in the unloaded state, and there is no class object associated with the class. At this point, there is a data structure that can be used to refer to the class. This data structure is a link time global with the name `<className>ClassData` and is almost always exported by name in its DLL (in the CFM sense). At some point when the class is used or asked for, the class object will be constructed.

The class can have “static” or “dynamic” status. By default the class is given “static” status. As an example of static use, the class can be loaded because an instance object was desired using the kernel operation `somNewObject` which internally uses the `<className>ClassData` global to refer to the class. If the class object (or an instance object) is obtained through any by name (as in runtime string) or by somId lookup services in the runtime kernel, the class is given “dynamic” status. For example `somGetDynamicClassReference` and `somFindClass` obtain dynamic class object references.

Static Class Status

Static class references are class object pointers that are obtained via kernel services (e.g. `somNewObject`, `somNewClassReference`, `SOMObject::somGetClass`) that take a statically obtained class data reference (i.e.: the address of the exported `<className>ClassData` structure). They do not involve CFM to either search for or load class code from the disk. The code for a statically referenced class is guaranteed to be in the closure for the code doing the reference, and thus to have been already loaded by CFM. The class code for these references is guaranteed be loaded for at least as long as the code that has the static class data reference in the first place. The operations that obtain a class object reference statically have higher performance and less overhead than ones in the dynamic reference category. However they are less secure in that static references are not designed to be able to outlive the lifetime of the code that had the initial static class data reference. By contrast, the lifetime of a dynamic reference is not limited. Thus, statically obtained references are intended for local and short term use of the class objects where the locality applies to the code involved in the CFM closure. When a class' reference count goes to zero, if the class has static status, then there is no CFM reference to release. The class should be considered to be unbuilt.

Dynamic Class Status

Dynamic class references are class object pointers that are obtained via runtime name or id class lookup services (e.g. `SOMClassMgr::somFindClass`, `somNewObjectByName`, `somGetDynamicClassReference`). They use CFM to search for and load a class from the disk and they always obtain a CFM reference to the class code. Thus having a dynamic class reference guarantees that the code for the class and all its inheritance hierarchy will not be unloaded until the last reference is released. If a class' reference count goes to zero and the class has dynamic status, then the CFM reference to the class' code is released.

Using Static vs. Dynamic

Once dynamic operations are used to obtain a reference to a class, all references to the class then become dynamic references. This is to say that at such time as a dynamic reference to the class is requested, the kernel will obtain a reference to the class code with CFM in order that it can prevent class code unloading until at least such time as all references to the class through the kernel are released. Programmers should generally not need to worry about the difference between static and dynamic references, as the 2.0.7 kernel automatically obtains dynamic references whenever by name operations are used - those operations that could have loaded new code from the disk. However, interfaces should not be designed to pass static class object references outside code in the CFM closure or load set that created them. If an interface chooses to pass a static class object reference, it should be converted to dynamic first, via `SOMClass::somMakeDynamicClassReference`. This will ensure that the reference is safe to use outside the creating closure.

Note: this applies to instance objects as well; if the instance objects of a given class are to be handed outside the closure they were created from, the class should have dynamic status.

SOMobjects™ for Mac OS and CFM Interaction

The kernel uses CFM's services to locate class DLLs, to load or lock class code, to release class code, and also to be informed when class code is being unloaded.

When the kernel searches for a class by name or id, it uses CFM's `GetSharedLibrary` call to locate the library, passing in the exact name supplied. Provided that CFM can locate a library with such a `'cfrg'` name, then the kernel will search for the `<className>ClassData` export in that DLL. If the export is found then it is used to bring up the desired class.

Code fragments not in the default search path or which do not export the `<className>ClassData` global will be inaccessible to the kernel unless they are first loaded in by some other means and then accessed in such a way as to cause their class(es) to be built. Classes will automatically become known to the kernel if a code fragment uses a class internally or if it had a `somNewClass` call in its CFM init routine which will then be used to satisfy the request.

The kernel API may be used to load and unload code fragments according to default CFM `GetSharedLibrary` rules. It is also both possible and reasonable to load a class or class containing DLL via CFM directly. The following example uses CFM to load a specific code fragment which makes its presence known to the kernel and is accessible through kernel APIs until it is unloaded.

```
err = GetDiskFragment ( ... , kLoadLib, & theCFMConnectionId );
theId = somIdFromString ( "ModuleName::ClassName" );
classObj = somGetDynamicClassReference
      ( theId, major, minor, NULL );
SOMFree ( theId );
... use ( classObj ); ...
CloseConnection ( & theCFMConnectionId );
somReleaseClassReference ( classObj );
```

Note: If the class code needs to outlive the CFM unload call, then a dynamic reference to the class should be obtained as in the above example, which uses `somGetDynamicClassReference`. Even if the programmer releases the direct CFM reference to the code, the code will not go away until the all references to the class are also released.

Thus, for each class in a given code fragment, a `'cfrg'` name or alias with the fully qualified class name should be present. In addition, the `<className>ClassData` global should be exported. In some cases, it may be desirable to announce the class to the kernel via a `somNewClass` call in the DLL's CFM init routine. This can be done in addition to, or in place of the export and the `'cfrg'` name. Using the `somNewClass` call in the init routine of the DLL can be useful if the DLL might be loaded from a non-standard search path location via CFM directly instead of using the kernel. When using the `somNewClass` call in the init routine, the export (and `'cfrg'` name or alias) is still recommended, since it is necessary for other operations such as subclassing. There is no (longer a) requirement for a matching unregister call in the termination routine for the DLL.

SOMobjects™ for Mac OS class libraries that export all their classes via CFM and provide a 'cfrg' entry for each class do not require an initialization (init), main, or termination (term) routine. This is now handled automatically by the kernel.

SOMobjects™ for Mac OS does not invoke any specified main routine when loading class libraries.

It is important not to obtain a reference to one's self and then expect it to be released in one's somUninit (destructor), since the outstanding reference means that the object will never be destructed. It is also important to not obtain a reference to one's own class object during the initialization routine and then expect to be able to release it in the termination (term) routine for class' DLL, since if the class is considered dynamically loaded, the kernel will never release its CFM reference due to the outstanding reference obtained to one's own class.

The SOMClassMgr

The class manager provides facilities for dynamically loading and unloading code and is documented in the IBM Users Guide. It is expected that direct access to the class manager object will not be needed since the kernel provides higher level functions to accomplish the same purpose. For maximum code portability, use of the kernel functions is recommended.

To access the global class manager object, use the function `somGetClassManagerReference` which returns a duplicated reference to the kernel global object. When finished it should be released with `somReleaseClassManagerReference`.

To capture a snapshot of currently registered classes, allocate an object of type `SOMRegisteredClassList`. The object returned will contain a complete thread-safe list of all classes registered at the time of the call.

Subclassing of the class manager is not recommended in a component environment such as the Macintosh.

Customization of the System Object Model

Chapter 5 of the IBM Users Guide describes how to customize the System Object Model services. Since on the Macintosh, the kernel initializes itself during its CFM initialization routine, the timing for installing these customized services cannot be done as the first part of “main.”

The SOMobjects standard API allows for customization of several aspects, namely, memory management, class library loading, error condition handling, debugging and diagnostic output handling, and class management. Each of these areas (except class management) are customized via kernel-exported data items, which are pointers to functions.

Error handling

The System Object Model’s default error handling is to call its process termination routine on errors and to ignore warnings. For Applications, `ExitToShell` is called on fatal errors. This behavior can be overridden using the mechanisms described in the IBM Users Guide.

Examples of fatal errors include:

- Invoking a method on a null object pointer.
- Invoking a method with an object of the wrong class.

The kernel does not guarantee that such errors will always be detected.

Output and Diagnostics

The default diagnostic output is implemented by reporting diagnostic information to a file in the current directory called “SOM.output.” The file is opened each time a CR-terminated line of text is written to the output routine. This causes the file to contain complete debugging information even if the program crashes, and also allows viewing and editing of Output while SOMobjects™ for Mac OS is in operation. Output is written for each fatal error, or when calls are made to somPrintf. This behavior can be overridden using the mechanisms described in the IBM Users Guide.
Use of `somPrintf` requires the routine `vsprintf` from the standard C library. In order to reduce the size of the kernel, and considering the fact that `somPrintf` is primarily a debugging facility, the kernel does not link in the static version of `vsprintf`. Instead, the kernel dynamically loads the shared standard C library the first time that `somPrintf` is called.

Custom Memory Management

The memory management routines for SOMobjects™ for Mac OS are invoked by the standard procedure pointers `SOMCalloc`, `SOMFree`, `SOMMalloc`, and `SOMRealloc`. The implementation uses the Macintosh Memory manager, making `NewPtr` and `DisposePtr` calls in the application's heap. Replacement of these globals is not recommended on the Macintosh, due to thread safety issues as well as the possibility of invoking incompatible memory managers between allocation and freeing.

Loader Customization

The System Object Model provides access to the global procedure pointers SOMLoadModule and SOMDeleteModule as a means of modifying the default behavior of the CFM based dynamic class management. Use of these globals is not recommended. If the built in behaviors of the kernel and class manager objects are insufficient to dynamically load classes you are interested in, see the section on "SOMobjects™ for Mac OS and CFM" for information on how to use the CFM directly for this purpose.

If you really must…

If you need to replace the default memory manager, this must be done before the kernel initializes itself and before any clients call it. Since the kernel may be used by the initialization routine of a class library or client, getting in front of all other callers is not possible without a new mechanism.

This mechanism is as follows: the kernel imports the following routines from a library with the same name, e.g., `SOMCustomMemoryMgr`, `SOMCustomErrorMgr` and `SOMCustomOutputMgr`. In the kernel, the imports are marked with both "weak" and "init-before." Therefore, if a library with the name, say, `SOMCustomMemoryMgr`, is in the path of an application that uses SOMobjects™ for Mac OS, this library will be loaded and both its initialization routine (if present) and its `SOMCustomMemoryMgr` export routine (also optional) will be run during the initialization of the kernel. Either the initialization routine (recommended), or the `SOMCustomMemoryMgr` export routine should contain code to swap the default services, using the normal facilities (switching the function pointers). All the routines are invoked before `somEnvironmentNew` initializes the kernel.

Building Class Libraries and Clients

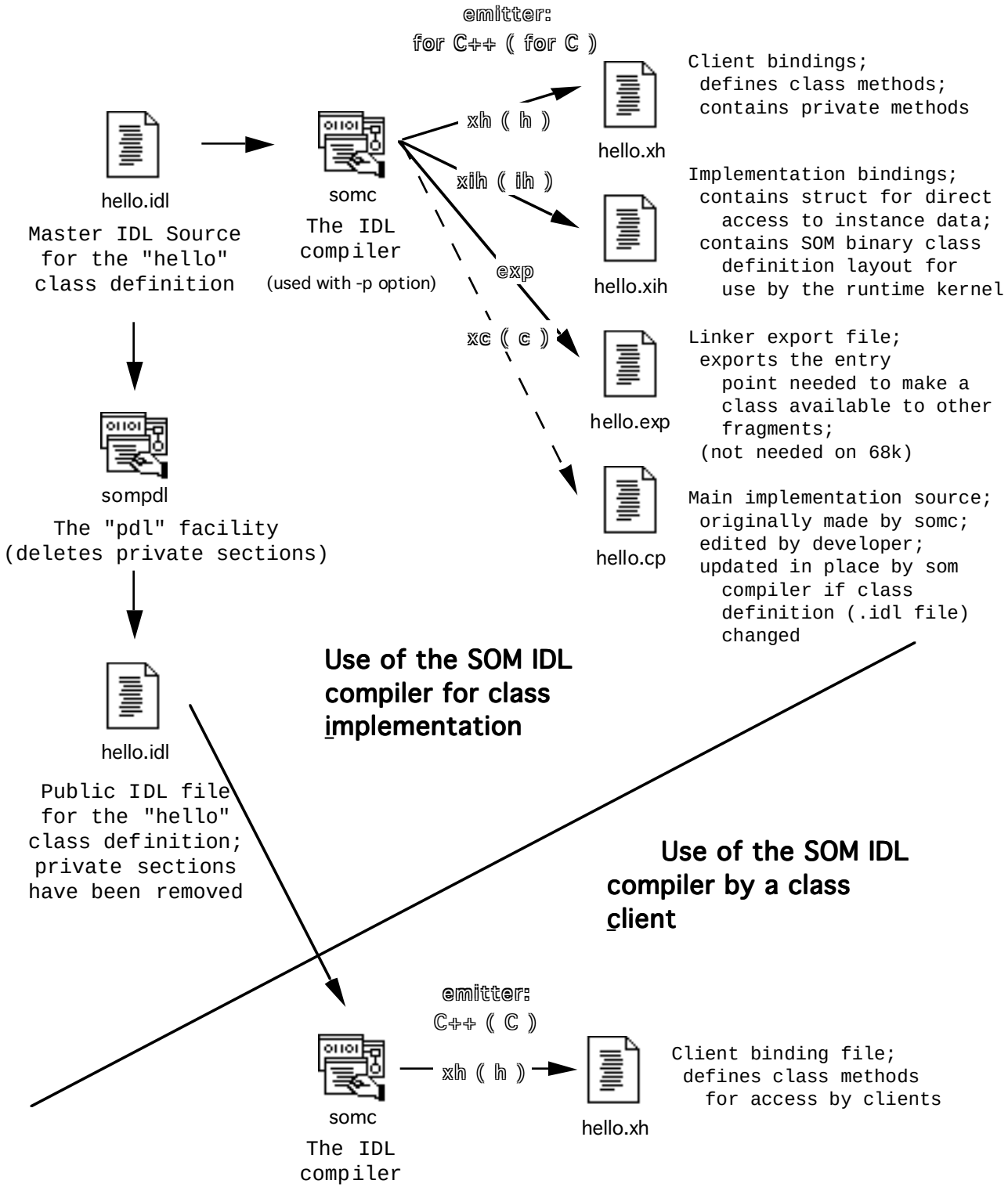
Overview

Chapter 2 of the IBM Users Guide provides examples of building classes and clients, both in C and C++. The interface to a new class must be defined in the Interface Definition Language, a language bearing a familial resemblance to C++. The .idl file is then compiled with the `somc` compiler, with appropriate emitters named on the command line. The choice of emitters depends on whether C or C++ is to be used for the source code of the client application and for the source code of the class implementation; these choices being made independently. It is possible to have more than one class interface defined in a single .idl file, but this practice is generally not recommended.

Compilation of .idl files during development typically results in two header files and a skeletal implementation file (the IBM documentation calls it a template file). The two header files, `<fileStem>` with `.h` and `.ih` suffixes for C, and`<fileStem>` with `.xh` and `.xih` suffixes for C++, are included when compiling the class implementation. The skeletal implementation file, `<fileStem>.c` for C and `<fileStem>.cp` for C++, contains the appropriate `#include` directives and skeletal functions corresponding to each declared class method. The programmer then has to fill in the implementation with his or her own code. At various points during class development, after changes are made to the class and method definitions in the .idl file (e.g.: adding a new method, or adding/removing or changing a parameter in a method), the IDL compiler can be re-run on the existing (no longer skeletal) implementation file, which keeps the implementation file in sync. The sompdl tool is used to produce a public version of the .idl file which does not contain items which were marked in the original .idl file by enclosure within `#ifdef __PRIVATE__ … #endif` (see IBM Users Guide, page 4-51) Typically the class binary and the public version of the .idl file are supplied to developers who will be clients of the class binary.

Compilation of .idl files by clients typically consists of taking the class developer supplied .idl file, and producing one of two header files, `<fileStem>.h` for C and `<fileStem>.xh` for C++ depending upon the language of the client's choice. Clients may choose their own `somc` compiler settings when compiling the client file (independent of the settings used by the implementor).

This illustration shows the use of the somc compiler and sompdl tool.



Building Fat Libraries

The following illustrative command lines are taken from the “hello” C++ Lib example. SOMobjects™ for Mac OS makes use of constant string data, requiring the use of the “-b3” option for SCpp and the “-roistext on” option to PPCLink.

```
somc -p -e xc,xih,xh,exp hello.idl

SCpp -model cfmflat -b3 hello.cp -I :
MrCpp hello.cp -o hello.cp.xcoff -I :

ILink -model cfmflat -state nouse -pad 0  ∂
      -xm sharedlibrary -fragName hello   ∂
      -o hellocp.tmp hello.cp.o           ∂
      "{CFM68KLibraries}"NuMacRuntime.o   ∂
      "{SharedLibraries}"somlib           ∂
      "{SharedLibraries}"StdCLib          ∂
      "{SharedLibraries}"InterfaceLib
MakeFlat hellocp.tmp -o hellocp.68k

PPCLink -roistext on -@export hello.exp   ∂
      -xm sharedlibrary -fragName hello   ∂
      -o hellocp.ppc hello.cp.xcoff       ∂
      "{PPCLibraries}"StdCRuntime.o       ∂
      "{PPCLibraries}"PPCCRuntime.o       ∂
      "{SharedLibraries}"somlib           ∂
      "{SharedLibraries}"StdCLib          ∂
      "{SharedLibraries}"InterfaceLib

Delete -i hellocp
MergeFragment hellocp.68k hellocp.ppc hellocp
```

Building a Fat Client Application

```
somc -e xh ::hello.idl

SCpp -model cfmseg -b3 main.cp -I :
MrCpp main.cp -o main.cp.xcoff -I :

ILink -model cfmseg -state nouse -pad 0   ∂
      -compact -o maincp.68k              ∂
      main.cp.o hellocp                   ∂
```

```
      "{CFM68KLibraries}"NuMacRuntime.o   ∂
      "{SharedLibraries}"somlib           ∂
      "{SharedLibraries}"StdCLib          ∂
      "{SharedLibraries}"InterfaceLib

PPCLink -roistext on -m __cplusstart      ∂
      -o maincp.ppc main.cp.xcoff hellocp ∂
      "{PPCLibraries}"StdCRuntime.o       ∂
      "{PPCLibraries}"PPCCRuntime.o       ∂
      "{PPCLibraries}"MrCPlusLib.o        ∂
      "{SharedLibraries}"somlib           ∂
      "{SharedLibraries}"StdCLib          ∂
      "{SharedLibraries}"InterfaceLib

Duplicate -y maincp.68k maincp
MergeFragment maincp.ppc maincp
```

Reference

This section documents the Apple recommended developer interface to the SOMobjects™ for Mac OS runtime kernel and classes. These functions have been designed to provide functionality with high performance.

There are functions and methods supported by the runtime kernel, documented in the IBM User's Guide that are not listed in the Apple recommended interface. These other functions and methods are present as system programming interfaces to support other subsystems (such as DSOM) and to provide cross platform backward compatibility. Use of functions and methods not in the recommended set may result in higher overhead and lower performance than desired.

Kernel API

Allocating and Releasing Object References

somNewObject - macro

This is a kernel operation that creates an object by class name identifier (not a string). It is provided to bypass the need to access the class object to create an instance object of the class. NULL is returned on failure to create the class, on version number failures, and on out of memory failures.

somNewVersionedObject - macro

This kernel operation is the same as somNewObject, except that this operation allows specification of exact version numbers instead of using the class version numbers that were current at compile time of the source code. NULL is returned on failure to create the class, on version number failures, and on out of memory failures.

somNewObjectById - function

This kernel operation obtains a dynamic reference to the class which then released. Before release, an instance object of the class is created. If obtaining the dynamic class reference fails, or creating the instance object fails, NULL is returned.

somNewObjectByName - function

This kernel operation is the same as somNewObjectById, but takes a zero terminated string instead of a somId for the class name.

somReleaseObjectReference - function

This kernel operation releases an object reference. If it turns out to be the last outstanding reference to the object, then the standard delete operations are invoked.

Class Object Access

somNewClassReference - macro

This kernel operation takes a class name as an identifier (a token, not a string) and returns a class object or NULL on failure. The version requested is the current interface version at build time. The returned class object must be released via somReleaseClassReference.

somNewVersionedClassReference - macro

This kernel operation is the same as somNewClassReference, but allows for specification of version checking with numbers other than the compile time interface version. This is useful when using interfaces which are more modern than absolutely required.

somGetDynamicClassReference - macro

This kernel operation obtains a dynamic reference to a class. The returned class object must be released via somReleaseClassReference. You can choose to hold onto the reference to the class object until such time as you want the class DLL to go away, which will prevent thrashing if you are going to use the class repeatedly. Or, you can release the class code reference immediately, and leave the class to be unloaded when the last outstanding object of the class is freed.

somReleaseClassReference - macro

This kernel operation releases a static or dynamically obtained class reference. If at some point the class obtained “dynamic” load status, then when the use count goes to zero (using this release mechanism) the classes code will be released. If releasing the code actually causes the code to be unloaded, then the class object will be freed.

somNewClass - macro

This function is used to announce the availability of a class to the kernel. It is generally not required, though can be useful in helping locate the class. For example, if the DLL is not in the standard search path, then the kernel will not be able to load the library via standard services; the library would have to be loaded via CFM directly. In such a case, the somNewClass call would identify that a new class (that the kernel could not otherwise find) had been loaded.

somKillClass - macro

This function is used to announce the unavailability of a class to the kernel. If the classes code and data is loaded and unloaded via CFM, then somKillClass is not necessary and should not be used. However to illustrate its use, it could be called from within a DLL’s termination routine to announce the unavailability of the class. It is designed for use with classes that are not loaded using CFM. One such possibility could be a SmallTalk system which may dynamically create a SOMObject based class without loading it from the disk.

Class Manager Access

somGetClassManagerReference - function

Returns a duplicated reference to the kernel class manager object which must be released via somReleaseClassManagerReference when no longer needed.

somReleaseClassManagerReference - macro

Releases the reference to the kernel class manager obtained from somGetClassManagerReference.

Ids

corbastring - type

A null terminated character string corresponding to the IDL data type `string`.

somId - type

An intermediate representation of a string which can represent a class or method name. somIds are not SOMObjects. somIds have allocated storage that should be freed via SOMFree.

somIdFromString - function

Creates a somId from a string. The somId can be passed to various other kernel operations. The somId must be freed by the caller using the function pointed to by SOMFree.

somMakeStringFromId - function

Returns the string associated with the id when it was created. The returned string should be freed by the caller using the function pointed to by SOMFree.

somKernelId - type

This type is introduced to differentiate between regular somIds which are owned by the caller (as handed out by somIdFromString) and kernel ids which are owned by the kernel (as handed out by somGetMethodDescriptor). The change to use somKernelIds for the kernel owned return values allows for the introduction of functions that translate the kernel ids into regular somIds for consistency.

somConvertAndFreeKernelId - function

This kernel operation receives a kernel id and converts it to a regular id. Be sure to free the id returned when need for it is through.

somFreeKernelId - function

This kernel operation frees a kernel id. This is called when a method or function is invoked that returns a kernel id that is not wanted.

Environments

SOM_InitEnvironment - macro

Initializes a local environment structure (clears it). For use with stack based environment structures.

SOM_UninitEnvironment - macro

Uninitializes a local environment structure (i.e.: frees any exception data, if set). For use with stack based environment structures.

SOM_CreateLocalEnvironment - function

Allocates a local environment structure and initializes it (clears it).

SOM_DestroyLocalEnvironment - function

Uninitializes an environment structure (i.e.: frees any exception data), and then releases the environment structure.

Exception Handling

somExceptionFree - function

Releases the storage associated with an exception referenced from an environment variable. Used during handling of exception returns.

somExceptionId - function

Returns the name of the exception given an environment variable. Used during handling of exception returns.

somExceptionValue - function

Returns the value of the exception given an environment variable. Used during handling of exception returns.

somSetException - function

Marks an exception in an environment structure. Use to indicate an exception.

Memory

SOMMalloc - pointer to function

The function pointed to by SOMMalloc allocates free storage and returns it. Note that SOMMalloc is task safe. Returns null when not enough space. Use the function pointed to by SOMFree to dispose of the memory.

SOMFree - pointer to function

The function is for releasing storage that was allocated via SOMMalloc, SOMCalloc or SOMRealloc. It is also for releasing storage of non-object data returned by pointers from kernel and other functions and methods. Note that SOMFree is task safe.

SOMCalloc - pointer to function

The function pointed to by SOMCalloc allocates free storage and returns it after clearing the storage. Note that SOMCalloc is task safe. Returns null when not enough space. Use the function pointed to by SOMFree to dispose of the memory.

SOMRealloc - pointer to function

The function pointed to by SOMRealloc re-allocates free storage originally allocated by SOMMalloc, SOMCalloc or SOMRealloc, and returns it. Note that SOMRealloc is task safe. Returns null when not enough space. Use the function pointed to by SOMFree to dispose of the memory.

SOMObject API

somInit - method

This method is the object's constructor, automatically invoked by most object allocation services. Override this method to initialize any instance data introduced by the overriding sub-class. Note that all instance data for an new object is automatically set to zero prior to its construction, removing the need for overriding this method in many cases. This method cannot be invoked on an object repeatedly.

somUninit - method

Destructs the object. Override somUninit to release storage or references upon object destruction. Note that this method should not be invoked directly unless the destructing objects created via somRenew (i.e.: auto objects). Prefer the use of somDestructAutoObject for destructing auto objects. Look for Apple documentation on building SOMobjects runtime compilers.

somDuplicateReference - method

This kernel operation acquires a reference to the given object. A new object pointer is returned, and should be taken as the new reference.

somCompareReference - method

This kernel operation compares two object references for object identity. It should be used to determine object identity instead of direct pointer compares for equality.

somRelease - method

Use this call to release a reference to any object or class. If the reference count goes to zero, the object will be freed provided that its somCanDelete call returns true. Note the currently, use of somReleaseClassReference and somReleaseObjectReference are preferred for source portability to IBM platforms; it is easier to provide simple source compatibility for kernel functions than for kernel methods.

somFree - method

If the object has outstanding references then one of the references is released and the object is not destructed or deallocated. If the object has only one outstanding reference (i.e.: its ref. count goes from 1 to 0) then the method somCanDelete is invoked to determine if the object should be deleted. The base implementation of somCanDelete always returns true. If somCanDelete returns true, then the object is destructed (via somUninit) and deallocated using the class' deallocation operation. Note that if the object's reference count is zero (or less) on entry to somFree, then the somCanDelete test is bypassed and the object is destructed and deallocated.

somCanDelete - method

This kernel operation is provided to allow classes to prevent the destruction of an object. When the use count of an object goes to zero, then prior to the destruction of the object (the invocation of somUninit) the somCanDelete message is invoked. If the somCanDelete message returns true, the object will undergo normal destruction. Otherwise, the object will be left undestructed.

somGetClass - method

Returns a reference to a class object given the receiving instance object. This class object reference does not mark the class as dynamic. The class object reference must be released via somReleaseClassReference by the caller.

somGetClassName - method

Invokes somGetName on the class object. Do not attempt to free the returned string.

somIsA - method

Tests whether the receiving object is an instance of the indicated class or of one of its sub-classes.

somIsInstanceOf - method

Tests whether the receiving object is an instance of the indicated class.

SOMClass API

An important rule for meta-class programming is that no methods introduced by SOMClass should ever be overridden. While this limits the utility of meta-class programming, it guarantees correct operation. Special class frameworks may be available to alleviate this restriction.

somNew - method

Used to allocate instance objects of the receiving class. Invokes the object's constructor, somInit.

somMakeDynamicClassReference - method

Use this call to convert a static class reference into a dynamic class reference (i.e.: to give the class dynamic status). This is used if the interface to the class required handing out (statically obtained) class object references to arbitrary clients.

somGetInstanceSize - method

Returns the total size of an instance of this class.

somGetName - method

Returns the name of the class. The return value is a pointer to a constant zero terminated string which cannot be freed.

somClassReady - method

Invoked by the runtime kernel to let the class know that it has been initialized and to give it a chance to perform any additional initialization before it is used.

somGetVersionNumbers - method

Returns the major and minor version numbers for this class.

somDescendedFrom - method

Used to verify inheritance. Tests whether a specified class is derived from the receiving class.

SOMClassMgr API

somClassFromId - method

Returns a reference to a class object given a somId. Searches for the class locally only; will not invoke CFM to load the class from the disk. The class referenced by the class object returned is marked as dynamic. The class object reference must be released via somReleaseClassReference by the caller. Returns null on failure to locate the specified class. Note that the function somNewObjectByName or somNewObjectById simplify creation of objects by name.

somFindClass - method

Returns a reference to a class object given a somId. Searches for the class locally first, then CFM to load the class from the disk if necessary. The class referenced by the class object returned is marked as dynamic. The class object reference must be released via somReleaseClassReference by the caller. Returns null on failure to locate the specified class. Note that the function somNewObjectByName or somNewObjectById simplify creation of objects by name.

somFindClsInFile - method

Returns a reference to a class object given a somId in the specified “file.” The “file” is taken as a DLL name (i.e.: 'cfrg' entry name). Searches for the class locally first, then CFM to load the class from the disk if necessary. The class referenced by the class object returned is marked as dynamic. The class object reference must be released via somReleaseClassReference by the caller. Returns null on failure to locate the specified class. Note that the function somNewObjectByName or somNewObjectById simplify creation of objects by name.

somUnregisterClass - method

Releases a reference to a class object. Interchangeable in operations with the kernel function somReleaseClassReference. Use somReleaseClassReference instead.

SOMRegisteredClassList API

By allocating an object of this class, a snapshot of the classes currently registered with the kernel may be obtained. This object provides task safe access to the list of classes it contains. The object must be released before any of the classes it refers to will be allowed to unload.

somNewIndexedClassReference - method

Returns a duplicated reference to the class object represented by a zero based index. Returns NULL if the index is out of range.

somNumRegisteredClasses - method

Returns the number of classes contained within the list.

Items no Longer Supported

SOMobjects 1.0 functions marked as being obsolete in SOMobjects 2.0 are no longer supported in SOMobjects™ for Mac OS.

SOMObject

somDispatchL, somDispatchV, somDispatchD

See IBM SOMobjects documentation for details.

SOMClass

somInitClass, somOverrideMtab, somAddStaticMethod, somInitMIClass

The somOverrideMtab method is being superseded by a new proxy class creation facility. Use somNewClass to create classes instead of somAddStaticMethod. Override somClassReady rather than somInitMIClass.

Differences between the Mac OS and IBM implementations

SOMobjects™ for Mac OS was designed to provide high source compatibility with IBM's SOMobjects™ implementations. There are some differences to be aware of, however. In particular, the Macintosh provides backward compatibility to the 2.0 level, whereas IBM provides backward compatibility to the 1.0 level.

SOMobjects™ for Mac OS Additions

Release Functions

We have introduced several "release" kernel functions. These functions release references to class objects, class implementations, and other objects like the class manager object.

- `somReleaseClassReference` - for releasing class objects and implementations
- `somReleaseObjectReference` - for releasing user objects
- `somReleaseClassManagerReference` - for releasing a class manager object

We have restricted direct data access to class objects and to the class manager object. Accessor APIs have been provided in their place. This was necessary in order to track object references. The new APIs require balancing release calls.

- Direct access to `<className>ClassData.classObject` is replaced with `somNewClassReference (<className>)`.
- Direct access to the class manager object via `SOMClassMgrObject` is replaced with `somGetClassMgrReference ()`.

Object Allocation

We provide several simplifying facilities to allocate objects without involving class objects or the class manager directly. These are:

- `somNewObject (<className>)`
- `somNewVersionedObject (<className>, majorVersion, minorVersion)`
- `somNewObjectByName ("className")`
- `somNewObjectById (classId)`

We provide simplified dynamic class object facilities without directly involving the class manager.

- `somGetDynamicClassReference (classId, …)`

Determining List of Loaded Classes

In order to capture the list of loaded classes, allocate an object of the class `SOMRegisteredClassList`. Objects of this type provide a task-safe snapshot of the class list at the time they are created. Classes in this list are prevented from being unloaded until the `SOMRegisteredClassList` object is released.

somIds

To differentiate between kernel and client owned somIds, we have introduced a new type and associated APIs. Furthermore a new function has been added which aids in the efficient handling of somIds.

- `somKernelId` - a new type designating kernel owned ids.
- `somFreeKernelId`
- `somConvertAndFreeKernelId` - converts `somKernelIds` into `somIds` and allows the kernel to dispose of the its id.
- `somMakeStringFromId` - a high performance replacement for `somStringFromId`.

Changes and Removals

Object References

All of the interfaces which return pointers to class objects have been modified to return references, which must later be released with the releasing interfaces as described. Specific methods changed were:

- `SOMObject::somGetClass` - returns a class object reference which must later be released
- `SOMClassMgr::somClassFromId` - " "
- `SOMClassMgr::somFindClass` - " "
- `SOMClassMgr::somFindClassInFile` - " "
- `SOMClassMgr::somLoadClassFile` - " " (also designated as protected)
- `SOMClassMgr::somUnregisterClass` - now releases a reference to a class rather than unconditionally tearing it down.

Bindings Interface

The binding level data structure `<className>CClassData` has been merged into the `<className>ClassData` structure in the Mac OS version. These data structures are primarily used by the language and platform specific bindings. It is recommended that your code avoid directly accessing them where possible.

The macro `_<ClassName>`, which accessed the `<className>ClassData.classObject` field is no longer provided. Use `somNewClassReference (<className>)` instead.

Dual Usage Functions

Some functions have multiple purposes which overlap with other API functions. The following functions or function types have been changed to represent their primary functionality in order to reduce ambiguity:

- `somEnvironmentNew` initializes the kernel, but no longer returns the global class manager. The function `somGetClassManagerReference` provides that function.
- The function `<className>NewClass` is not provided by the Mac OS version bindings. Its primary use was to ensure that the class is ready to create instance objects, which is no longer necessary as this is handled automatically by the kernel. The second use was to access the class object, which is already provided by `somNewClassReference`.
- The object macro `SOM_GetClass` is no longer supported due to direct global access. Use the method `somGetClass` instead.

somIds

Functions and methods returning kernel owned ids in the form of a somId were changed to return instead a somKernelId. Affected were:

- `SOMClass::somGetMethodDescriptor`
- `SOMClass::somGetNthMethodInfo`

Some convenience macros are no longer supported due to performance requirements.

- SOM_StringFromId - should be replaced with the function somMakeStringFromId instead to obtain a `corbastring` which must then be freed with SOMFree.
- SOM_IdFromString - should be replaced with the function somIdFromString. The returned somId must then be freed with SOMFree.
- somUniqueKey - Use user defined function instead
- somRegisteredId - Use user defined function instead

Version 1.0 support

- `integer4` - should be replaced with `long`, -or- defined by the user if desired
- `zString` - should be replaced with `char*`.
- `string` - should be replaced with `corbastring`.

Other Differences

- `SOMObject::somFree` - One should override somUninit or somCanDelete instead. Calling through to the parent somFree can cause severe problems in multiple inheritance. Objects can be created without ever using the somNew / somFree memory allocation services. Because of these and similar issues, somUninit should be used to receive messages about object destruction. Method somCanDelete can be used suppress object destruction.
- `somIsObj` - This function is described as being “fail-safe” in IBM's documentation. It is NOT fail-safe on System 7. Note that since the kernel does not track all instance objects, the function somIsObj cannot with certainty actually determine if the item in question is an actual SOMObject, but rather only if it looks like a valid SOMObject.

Cross-Platform Portability

Tips on Writing Portable Code

The majority of functionality obtainable from the SOMobjects Toolkit is common to all implementations, so most code takes care of itself. Areas to watch out for are obvious; functions not available on all platforms will be non-portable.

The SOMobjects™ for Mac OS APIs are designed with portability in mind. Where not directly available from IBM kernels, most are accommodated by compiling with the appropriate preprocessor define enabled.

Do’s

- Use the modern APIs
- Release all references you own
- Free all somIds

Don'ts

- Access globals directly
- Rely upon the somId interfaces unnecessarily
- Invoke methods described as "protected"
- Invoke somInit or somUninit directly on objects
- Override methods which could cause dynamically loaded classes to unload
- Mix old and modern API code bases

Porting IBM Code to the Macintosh

By defining the preprocessor symbol `__IBM_TO_MAC__` to 1 when compiling C/C++ or IDL files, most IBM style API calls not included in the Macintosh API set can be accommodated.

Porting Macintosh Code to IBM

By defining the preprocessor symbol `__MAC_TO_IBM__` to 1 when compiling C/C++ or IDL files, most of the APIs ported in this document will be able to run under non Macintosh kernels.

At the present time the reference interface extensions to SOMObject will not port directly to non Macintosh platforms. We recommend that subclasses whose clients use these methods inherit from SOMRefObject (defined in "somref.idl") instead. On other platforms, it can be a real class. A reference implementation is provided in somref.c. Sources for both are provided as an appendix to this document.

Runtime Considerations

Thread Safety

The function somGetGlobalEnvironment is not Thread Manager thread-safe on System 7. When multiple threads may be an issue, use a local (stack based/automatic) environment variable initialized with SOM_InitEnvironment, and uninitialized with SOM_UninitEnvironment instead of using the somGetGlobalEnvironment function.

Note that Copland versions of the runtime kernel will be Task safe.

Notes and Cautions

Developers and clients of SOMObject based class libraries should be aware that, due to the mechanisms of the underlying runtime architecture, 68k class binaries and 68k clients will not run emulated on Power Macintosh computer systems. The preferred course of action is to always build and package both CFM-68K and Power Macintosh (native) code fragments in the same class library file. This dual packaging is called "fat." The document Building Programs for Macintosh with PowerPC on the MPW Pro disc describes how to build a fat application using classic 68k code. The process for CFM-68K is similar. See the CFM-68K MODApp example for an illustration of the process.

Since the kernel is implemented as a standard application level shared library, each application which uses it is completely independent from all others. Only the lower level Code Fragment Manager is aware when two processes are using the same class library code fragment. This allows each application to have a completely independent and separate class hierarchy and name space.

Multiple inheritance in the System Object Model is implemented using a technique similar to "virtual" inheritance in C++. A base class will never be represented more than once in any hierarchy. An undesirable side effect of virtual inheritance can be seen in the following example of parent call-through:

```
interface SOMObject { … somPrintSelf () … };
interface B : SOMObject { … somPrintSelf : override … };
interface C : SOMObject {  … somPrintSelf : override … };
interface D : B, C {  … somPrintSelf : override … };

D_somPrintSelf ()
{
```

```
        D_parent_B_somPrintSelf ();
        D_parent_C_somPrintSelf ();
        /* D code here */
    }
```

The first parent call-through will call B's `somPrintSelf` method, which will call SOMObject's `somPrintSelf` method. The second parent call-through will call C's `somPrintSelf` method which in turn will call SOMObject's `somPrintSelf` for a second time.

While this behavior is relatively harmless for `somPrintSelf` it can be a serious problem for more significant methods. Note that the same problem exists for C++ when using virtual inheritance.

```
//
//    somref.idl
//    Copyright:  © 1995 by Apple Computer, Inc.
//                        All rights reserved.

#ifndef     somref_idl
#define     somref_idl

#include <somobj.idl>


interface SOMRefObject : SOMObject {

  SOMObject somDuplicateReference ();
  boolean somCompareReference ( in SOMObject anObject );
  SOMObject somRelease ();
  boolean somCanDelete ();

#ifdef __SOMIDL__
  implementation {
      releaseorder:
                  somDuplicateReference,
                  somCompareReference,
                  somRelease,
                  somCanDelete;

    callstyle = oidl;
    externalstem = somref;
    majorversion = 70;
    minorversion = 1;
    filestem = somref;
    dllname = "somref.dll";

    somRelease: nooverride;

  #ifdef __PRIVATE__
      long fRefCount;
    somFree: override;
    functionprefix = somro_;
  #endif /* __PRIVATE */
  };
#endif /* __SOMIDL__ */
};


#endif /* somref_idl */
```

```
/*
 *  somref.c
 *  Copyright:    © 1995 by Apple Computer, Inc.
 *                      All rights reserved.
 */


#define SOMRefObject_Class_Source
#include "somref.ih"

/* The following macro should be redefined to invoke system specific
      thread safe (atomic) increment functions.
      Here it is not thread safe. */

#define ATOMICADDTOMEMORY(mem,x) \
      (*(mem)+=(x))

SOM_Scope SOMObject* SOMLINK somro_somDuplicateReference
      (SOMRefObject *somSelf)
{
      SOMRefObjectData *somThis = SOMRefObjectGetData (somSelf);
      ATOMICADDTOMEMORY ( & somThis -> fRefCount, 1 );
      return somSelf;
}

SOM_Scope boolean SOMLINK somro_somCompareReference
      (SOMRefObject *somSelf, SOMObject* anObject)
{
      return ((SOMObject*) somSelf) == anObject;
}

SOM_Scope SOMObject* SOMLINK somro_somRelease (SOMRefObject *somSelf)
{
      SOMRefObjectData *somThis = SOMRefObjectGetData ( somSelf );
      if ( ATOMICADDTOMEMORY ( & somThis -> fRefCount, -1 ) == 0 ) {
            if ( _somCanDelete ( somSelf ) )
                  SOMRefObject_parent_SOMObject_somFree ( somSelf );
      }
      return 0;
}

SOM_Scope void SOMLINK somro_somFree (SOMRefObject *somSelf)
{
      SOMRefObjectData *somThis = SOMRefObjectGetData ( somSelf );
      if ( ATOMICADDTOMEMORY ( & somThis -> fRefCount, -1 ) <= 0 ) {
            if ( somThis->fRefCount < 0 || _somCanDelete ( somSelf ) )
                  SOMRefObject_parent_SOMObject_somFree ( somSelf );
      }
      return 0;
}

SOM_Scope boolean SOMLINK somro_somCanDelete (SOMRefObject *somSelf)
{
      return 1;
}
```