Appendix A. Customer Support and Error Codes

Service and Technical Support for SOMobjects

This service and technical support information applies for:

- **SOMobjects Developer Toolkit, Version 2.1**
- **SOMobjects Workstation Enabler, Version 2.1**
- **SOMobjects Workgroup Enabler, Version 2.1**

Notes: Customers in European, Middle Eastern, and African Countries should refer to the separate Service Statement included with the product for service and technical support instructions for this product.

Customers in Canada and Asia Pacific Countries should refer to the Service Statement in the License Information Booklet for service and technical support instructions for this product.

You Must Register for Service

Defect service for this product is available through April 30, 1996, or six months after the general availability of a subsequent version of the product (or a product designated as a replacement product), whichever occurs earlier.

Register by providing your company name, address, phone number, Internet address (if applicable), contact person's name, phone and FAX numbers (include area code). This information can be sent via electronic mail as follows:

- IBM OS2BBS to userid: **WZ00178**

or

- Internet Commercial: **somreg@austin.ibm.com**

or

- CompuServe: **GO IBMSOM**
  and then browse the News Flash for further registration information.

Within two working days of receipt of your registration, a service ID or password will be issued to you, allowing access to the defect forum and technical support forum.

Defect Support

Defect service for this product is available through April 30, 1996, or six months after the general availability of a subsequent version of the product (or a product designated as a replacement product), whichever occurs earlier.

Defect service is provided by the IBM SOMobjects Development personnel via the following Electronic Support Services:

- IBM OS/2 Bulletin Board System
  via IBM TalkLink Electronic Conferencing Service
- Internet Commercial Electronic Network
- CompuServe

The IBM SOMobjects Development personnel will monitor these Electronic Support Services between 9 a.m. and 6 p.m. CT, Monday through Friday, except holidays. Acknowledgement of receipt of Defect Report will be within 24 hours for SOMobjects RUNTIME defects and 72 hours for SOMobjects TOOLKIT defects, provided that the Defect Report is received by the SOMobjects Technical Support personnel during the time period of 9 a.m. to 6 p.m. CT, Monday through Friday.

Technical Support

Technical support service for this product is available for ninety (90) days after receipt of your service registration by SOMobjects Development personnel or until expiration of defect support, whichever occurs first.

Technical support service is provided by the IBM SOMobjects Development personnel via the following Electronic Support Services:

- IBM OS/2 Bulletin Board System
  via IBM TalkLink Electronic Conferencing Service
- Internet Commercial Electronic Network
- CompuServe

The IBM SOMobjects Development personnel will monitor these Electronic Support Services between 9 a.m. and 6 p.m. CT, Monday through Friday, except holidays. Questions will be answered in the order in which they are received. Extension of the technical support beyond the expiration date will be offered on a fee basis. Information regarding this offering will be provided on the service bulletin boards.

IBM OS/2 Bulletin Board System via TalkLink

The OS/2 Bulletin Board System (BBS) is implemented on the IBMLink facility. The OS/2 BBS is provided to all Workstation Technical Coordinators (WTSC) in corporate IBMLink accounts and all members of the OS/2 Developer's Assistance Program (DAP) who have access to IBMLink. You may contact your Technical Coordinator, if one has been identified by your company. If your company does not currently utilize IBMLink, you can subscribe to TalkLink by calling 1–800–547–1283 (USA).

How to use the IBM OS/2 Bulletin Board System (OS2BBS) via TalkLink for service and support for SOMobjects:

- To obtain **technical support** for non-defect "how-to" questions and answers:

  – Logon to IBM OS2BBS system from IBMLink Main Menu screen
  – Select "OS/2 Questions and Answer Bulletin Boards"
  – Select "SOMHOWTO" CFORUM

- To submit a suspected **defect report**:

  – Logon to IBM OS2BBS system from IBMLink Main Menu screen
  – Select "OS/2 Questions and Answer Bulletin Boards"
  – Select "SOMTKBUG" – if the suspected defect is with the SOM Toolkit
  – Select "SOMRTBUG" – if the suspected defect is with the SOM Runtime

Note: When submitting a suspected defect report, please provide the following information:

– Your Company name and address.
– Your name, phone and FAX numbers.
– The hardware platform – (PS/2 Model ____, RS/6000 Model ____, or other__________).
– Operating System and level –
  (OS/2 Version ____, AIX Version ____, or DOS/Windows Version____).
– System configuration (memory, communication protocol, etc.).
– SOMobjects Version _____ (and CSD level _____, if applicable).
– Complete description of the problem.
– Complete test case to reproduce the problem, if applicable (with a minimum amount of code/data).

Internet Commercial Electronic Network

How to use Internet for service and support for SOMobjects:

- To obtain **technical support**, for non-defect "how-to" questions and answers:
  - Via USENET Newsgroup at: **comp.unix.aix**
    Note: Include the word "SOM" in the subject line.
- To submit a suspected **defect report**:
  - Send EMAIL to: **sombug@austin.ibm.com**

Note: When submitting a suspected defect report, please provide the following information:

- Your Company name and address.
- Your name, phone and FAX numbers.
- Your Internet address.
- The hardware platform (PS/2 Model ____, RS/6000 Model ____, or other__________).
- Operating System and level –
  (OS/2 Version ____, AIX Version ____, or DOS/Windows Version____).
- System configuration (memory, communication protocol, etc.).
- SOMobjects Version _____ (and CSD level _____, if applicable).
- Complete description of the problem.
- Complete test case to reproduce the problem, if applicable (with a minimum amount of code/data).

CompuServe

How to use CompuServe for service and support for SOMobjects:

- From any CompuServe prompt, enter: **GO IBMSOM**

Note: When submitting a suspected defect report, please provide the following information:

- Your Company name and address.
- Your name, phone and FAX numbers.
- The hardware platform (PS/2 Model ____, RS/6000 Model ____, or other__________).
- Operating System and level –
  (OS/2 Version ____, AIX Version ____, or DOS/Windows Version____).
- System configuration (memory, communication protocol, etc.).
- SOMobjects Version _____ (and CSD level _____, if applicable).
- Complete description of the problem.
- Complete test case to reproduce the problem, if applicable (with a minimum amount of code/data).

If you are not currently a member of CompuServe, you can subscribe by calling (USA) 1–800–524–3388 and asking for Representative 239.

SOM Kernel Error Codes

Following are error codes with messages/explanations for the SOM kernel and the various frameworks of the SOMobjects Developer Toolkit.

Value	Symbolic Name and Description
20011	**SOMERROR_CCNullClass** The **somDescendedFrom** method was passed a null class argument.
20029	**SOMERROR_SompntOverflow** The internal buffer used in **somPrintf** overflowed.
20039	**SOMERROR_MethodNotFound** **somFindMethodOk** failed to find the indicated method.
20049	**SOMERROR_StaticMethodTableOverflow** A Method–table overflow occurred in **somAddStaticMethod**.
20059	**SOMERROR_DefaultMethod** The **somDefaultMethod** procedure was called; a defined method probably was not added before it was invoked.
20069	**SOMERROR_MissingMethod** The specified method was not defined on the target object.
20079	**SOMERROR_BadVersion** An attempt to load, create, or use a version of a class-object implementation is incompatible with the using program.
20089	**SOMERROR_NullId** The **SOM_CheckId** was given a null ID to check.
20099	**SOMERROR_OutOfMemory** Memory is exhausted.
20109	**SOMERROR_TestObjectFailure** The **somObjectTest** found problems with the object it was testing.
20119	**SOMERROR_FailedTest** The **somTest** detected a failure; generated only by test code.
20121	**SOMERROR_ClassNotFound** The **somFindClass** could not find the requested class.
20131	**SOMERROR_OldMethod** An old-style method name was used; change to an appropriate name.
20149	**SOMERROR_CouldNotStartup** The **somEnvironmentNew** failed to complete.
20159	**SOMERROR_NotRegistered** The **somUnloadClassFile** argument was not a registered class.
20169	**SOMERROR_BadOverride** The **somOverrideSMethod** was invoked for a method that was not defined in a parent class.
20179	**SOMERROR_NotImplementedYet** The method raising the error message is not implemented yet.
20189	**SOMERROR_MustOverride** The method raising the error message should have been overridden.

20199 **SOMERROR_BadArgument**
An argument to a core SOM method failed a validity test.

20219 **SOMERROR_NoParentClass**
During the creation of a class object, the parent class could not be found.

20229 **SOMERROR_NoMetaClass**
During the creation of a class object, the metaclass object could not be found.

DSOM Error Codes

The following table lists the error codes that may be encountered when using DSOM. (Obsolete messages have been removed, thus message numbers do not compose a full sequence.)

Value	Description
30001	**SOMDERROR_NoMemory** Memory is exhausted.
30002	**SOMDERROR_NotImplemented** Function or method has a null implementation.
30004	**SOMDERROR_IO** I/O error while accessing a file located in SOMDDIR.
30008	**SOMDERROR_HostAddress** Unable to retrieve local host address.
30019	**SOMDERROR_NoMessages** No messages available (and caller specified “no wait”).
30020	**SOMDERROR_UnknownAddress** Invalid client or server address.
30023	**SOMDERROR_CommTimeOut** Communications timeout. Make sure the DSOM daemon is running.
30026	**SOMDERROR_NoHostName** Unable to get host name.
30029	**SOMDERROR_BadEnvironment** Invalid Environment value in request message.
30031	**SOMDERROR_BadNVList** Invalid Named Value List (NVList).
30032	**SOMDERROR_BadFlag** Bad flag in NVList item.
30033	**SOMDERROR_BadLength** Bad length in NVList item.
30034	**SOMDERROR_BadObjref** Invalid object reference.
30036	**SOMDERROR_UnknownReposId** Attempt to use invalid Interface Repository identifier.
30037	**SOMDERROR_NVListAccess** Invalid NVList object in request message.
30038	**SOMDERROR_NVIndexError** Attempt to use an out-of-range NVList index.
30039	**SOMDERROR_SysTime** Error retrieving system time.
30041	**SOMDERROR_CouldNotStartProcess** System error: Unable to start a new process.
30042	**SOMDERROR_NoServerClass** No SOMDServer (sub)class specified for server implementation.
30043	**SOMDERROR_NoSOMDInit** SOM or DSOM has not been initialized.

30045 **SOMDERROR_NoImplDatabase**
Could not open Implementation Repository database.

30046 **SOMDERROR_ImplNotFound**
Implementation not found in implementation repository.

30047 **SOMDERROR_ClassNotFound**
Class not found in implementation repository.

30048 **SOMDERROR_ServerNotFound**
Server not found in **somdd**'s active server table.

30049 **SOMDERROR_ServerAlreadyExists**
Server already exists in **somdd**'s active server table.

30050 **SOMDERROR_ServerNotActive**
Server is not active.

30052 **SOMDERROR_ObjectNotFound**
Could not find desired object.

30053 **SOMDERROR_NoParentClass**
Unable to find / load parent class during proxy class creation.

30055 **SOMDERROR_BadTypeCode**
Invalid type code.

30056 **SOMDERROR_BadDescriptor**
Invalid method descriptor.

30059 **SOMDERROR_KeyNotFound**
Internal object key not found.

30060 **SOMDERROR_CtxInvalidPropName**
Illegal context property name.

30061 **SOMDERROR_CtxNoPropFound**
Could not find property name in context.

30062 **SOMDERROR_CtxStartScopeNotFound**
Could not find specified context start scope.

30063 **SOMDERROR_CtxAccess**
Error accessing context object.

30064 **SOMDERROR_CouldNotStartThread**
System error: Unable to start a new thread.

30065 **SOMDERROR_AccessDenied**
System error: Access to a system resource (file, queue, shared memory, etc.) is denied.

30066 **SOMDERROR_BadParm**
Invalid parameter supplied to an operating system call.

30072 **SOMDERROR_NoSpace**
System error: No space left on device.

30089 **SOMDERROR_WrongRefType**
Operation attempted on an object reference is incompatible with the reference type.

30090 **SOMDERROR_MustOverride**
This method has no default implementation and must be overridden.

30091 **SOMDERROR_NoSocketsClass**
Could not find/load Sockets class.

30092 **SOMDERROR_EManRegData**
Unable to register DSOM events with the Event Manager.

30093 **SOMDERROR_NoRemoteComm**
Remote communications is disabled (for Workstation DSOM).

30096 **SOMDERROR_GlobalAtomError**
On Windows only, an error occurred while adding a segment name to the Windows atom table.

30097 **SOMDERROR_NamedMemoryTableError**
On Windows only, an error occurred while creating or deleting a (named) shared memory segment.

30098 **SOMDERROR_WMQUIT**
On Windows only, indicates DSOM received a Windows WM_QUIT message. The developer of a server application should check for SOMDERROR_WMQUIT returned from method **execute_request_loop** and handle the error by cleaning up and exiting.

30105 **SOMDERROR_DuplicateImplEntry**
Implementation repository identifier already exists. Add wait time between 'regimpl' calls.

30106 **SOMDERROR_InvalidSOMSOCKETS**
SOMSOCKETS environment variable set incorrectly.

30107 **SOMDERROR_IRNotFound**
Interface Repository not found.

30108 **SOMDERROR_ClassNotInIR**
Attempt to create an object whose Class is not in the Interface Repository.

30110 **SOMDERROR_SocketError**
A communications socket error has occurred. Make sure the DSOM daemon is running.

30111 **SOMDERROR_PacketError**
A communications packet error has occurred.

30112 **SOMDERROR_Marshal**
A marshalling error has occurred.

30113 **SOMDERROR_NotProcessOwner**
On AIX only, the server cannot be killed because you are not the process owner.

30114 **SOMDERROR_ServerInactive**
The requested server is not running.

30115 **SOMDERROR_ServerDisabled**
The server has been disabled by the program servmgr.

XXXXX **SOMDERROR_OperatingSystem**
On AIX, this is the value of the C error variable "errno" defined in errno.h; on OS/2 and Windows, it is the DOS API return code.

Persistence Framework Error Codes

Methods of the Persistence Framework return a **sompException**, exception. The exception contains a **primary** and **secondary** value. The **primary** value will contain either SOMPERROR_SYSTEM_ERROR or SOMPERROR_FRAMEWORK_ERROR. The former is returned when the originating error comes from the underlying C library. The latter is returned when the error was detected within the Persistence Framework.

When the **primary** field is SOMPERROR_SYSTEM_ERROR, **secondary** will contain the actual error returned from the C library (the value normally found in the C variable "errno").

When the **primary** field is SOMPERROR_FRAMEWORK_ERROR, **secondary** will contain one of the following errors, defined in "somperr.idl":

Value	Description
10	**SOMPERROR_OBJ_ALREADY_REGISTERED** Attempted to register an object that was already registered. This is an internal error.
11	**SOMPERROR_COULD_NOT_FIND_DIR** The system ID assigner attempted to use the path specified in the environment variable SOMP_PERSIST to find the file containing the last assigned ID, and the path did not exist. Check the value of SOMP_PERSIST and make sure it points to a valid pathname.
12	**SOMPERROR_IOGROUP_EMPTY** One of the I/O Group Managers was asked to write an empty I/OGroup. This is an internal error.
13	**SOMPERROR_COULD_NOT_RESTORE_OBJ** A request was made to restore an object. The ID indicated a valid file, but the requested object was not in the file. This error may occur if a class object was not created prior to calling sompRestoreObject.
14	**SOMPERROR_OBJ_IS_NOT_PERSISTENT** Attempted to store an object whose class is not derived from SOMPPersistentObject. Only objects derived from SOMPPersistentObject can be stored.
15	**SOMPERROR_IOGROUP_NEWOBJ** Attempted to restore an object, but the appropriate class object could not be found. Users with this error should put their object in a dynamically loadable file or use their "<className>NewClass" function to create their class object before attempting to restore objects. If the objects are already in a DLL, ensure that: – The SOMInitModule function for the DLL executes the procedure <className>NewClass for the class which can not be restored. – On OS/2, ensure that the SOMInitModule entry point uses the "system" linkage convention (see samples). – The class definition in question has the "dllname" modifier. For example, if your dll was named foo.dll, you should have `dllname="foo.dll";` in your .idl file. – That you have updated the interface repository for your class. Assuming foo.idl, you can update the IR by running `"sc -sir -u foo.idl"` – Ensure environment variable SOMIR points to the correct interface repository file.

When all else fails, explicitly call the <className>NewClass procedure of your class before attempting to restore it.

16 **SOMPERROR_IOGROUP_NOTREAD**
Attempted to read an object whose I/O Group has never been read. This is an internal error.

17 **SOMPERROR_OBJ_IS_NOT_INITIALIZED**
Attempted to store an object which was not properly initialized. You need to initialize your persistent object with a persistent ID before storing it.

18 **SOMPERROR_PFW_INIT_FAILED**
Failure to instantiate either the SOMPPersistentStorageMgr or other internal classes. This is an internal error.

19 **SOMPERROR_ED_INVALID_DATA_TYPE**
Default Encoder/Decoder (SOMPAttrEncoderDecoder) attempted to read in some data, but could not recognize the data type. This could indicate a corrupted data file or an internal error.

20 **SOMPERROR_BAD_OBJECT_ID_STRING**
You have attempted to initialize a persistent ID with a string that is not of the appropriate syntax. The correct form of a persistent ID is:

```
<IOGroupMgrClassName>:<IOGroupName>:<GroupKey>
```

where:

`<IOGroupMgrClassName>` is the class name of an I/O Group Manager class. The framework supplies two: SOMPAscii and SOMPBinary,

`<IOGroupName>` is a name understandable to the I/O Group Manager Class. SOMPAscii and SOMPBinary expect this name to be a file name, and

`<GroupKey>` is a key number.

If this error occurs during restore, either the ID you have passed to the Persistent Storage Mgr is incorrect, or possibly the ID of an object embedded in the object you are attempting to restore is incorrect. The string ID read from storage may have been corrupted on disk.

21 **SOMPERROR_INVALID_OBJECT_TYPE**
One of the media interfaces attempted to determine the type of an object and couldn't. This could indicate a corrupted data file or an internal error.

22 **SOMPERROR_ENC_DEC_NOT_FOUND**
The class object of the encoder/decoder you are attempting to use can not be found. To determine what class the framework is attempting to find you can invoke the method

```
sompGetEncoderDecoderName
```

on the objects of the class you are attempting to save/restore.

This error is typically the result of specifying a user written encoder/decoder class (via either method sompSetEncoderDecoderName or sompSetClassLevelEncoderDecoderName) without first ensuring that its class object exists. Execute either the "NewClass" procedure of the encoder/decoder class or ensure that it is installed correctly in a DLL (see SOMPERROR_IOGROUP_NEWOBJ).

If the encoder/decoder class name is SOMPAttrEncodeDecoder, then there is a problem with the framework.

23 **SOMPERROR_IOGROUP_DOES_NOT_EXIST**
SOMPPersistentStorageMgr was trying to read or delete an object, but couldn't find the I/O Group. For SOMPAscii and SOMPBinary, this means the file could not be found. This error is the result of sompGroupExists method returning FALSE.

24 **SOMPERROR_OBJECT_NOT_FOUND**
The object could not be restored or deleted because it could not be found. Ensure that the SOMP_PERSIST environment variable is set the same as when you stored the object.

If the file appears to be there but you still can't restore, perhaps there is an upper/lower case difference in the name of the file. The framework is case sensitive.

25 **SOMPERROR_INTERFACE_NOT_FOUND**
One of the I/O Group Managers was attempting to copy a group and could not find the appropriate Media Interface. This is an internal error.

26 **SOMPERROR_NOT_STORING_OBJECTS**
sompAddObjectToWriteSet has been called; however, because the SOMPPersistentStorageMgr is not currently storing objects, there is no write set.

The sompAddObjectToWriteSet method is intended to be used by encoder/decoder objects which are run during a save/restore.

27 **SOMPERROR_POINTER_NOT_REGISTERED**
This is an internal error.

28 **SOMPERROR_MEDIA_FORMAT_ERROR**
Attempt was made to read a file with a particular I/O Group Mgr, but the file was incompatible with the requested I/O Group Mgr. Make sure the I/O Group Mgr specified in the persistent ID string is compatible with the I/O Group Mgr that wrote the file.

This error may also occur if you attempt to restore an object from a file produced by an aborted store. If the store attempt failed, the file may have been left in an invalid format.

29 **SOMPERROR_BAD_IOGROUP_MANAGER**
Attempted to restore an object with a nonexistent I/O Group Mgr. Check the name of the I/O Group Mgr in the Persistent ID string.

The I/O Group Mgr class name is the first part of a peristent ID. The class object of the I/O Group Mgr must exist prior to invoking store/restore requests to it. Ensure that you have either called the <className>NewClass procedure of your I/O Group Mgr class or have properly installed your class into a dynamically loadable library (see SOMPERROR_IOGROUP_NEWOBJ).

30 **SOMPERROR_NOT_RESTORING_OBJECTS**
sompAddIdToReadSet has been called but since the PersistentStorageMgr is not currently restoring objects, there is no read set.

The sompAddIdToReadSet method is intended to be used by encoder/decoder objects which are run during a save/restore.

31 **SOMPERROR_ED_AGGREGATE_TYPE_CHG**
An object has been redefined in a way that one of its aggregate data types is no longer compatible with its stored data. Typically this means that either

- A new member has been added to a structure.
- A sequence's type has been changed (i.e. what had been a sequence of reals is now a sequence of strings).
- An array's type has been changed (i.e. what had been an array of reals is now an array of strings.

32 **SOMPERROR_ED_ATTR_NOT_DEFINED**
An object has been redefined such that one of its attributes is no longer valid. During the restore of an object, data for an nonexistent attribute was found. The restore has been aborted.

33 **SOMPERROR_ED_TC_RESTORE_FAILED**
An attempt to read a type code failed. This could indicate corrupted data file or an internal error.

34 **SOMPERROR_ED_TYPECHG**
The definition for an attribute has changed its type. During the restore of an object, the data type of an attribute has been found to be different than stored data type. The restore has been aborted.

35 **SOMPERROR_ED_TYPE_SIZE_CHG**
The definition for a sequence has changed such that the amount of data stored in the sequence exceeds the new maximum.

36 **SOMPERROR_ED_UNSUPPORTED_TYPE**
Attempted to store a non supported type code. The type of the data was not a recognized CORBA data type.

37 **SOMPERROR_ED_CLASS_NOT_DEFINED**
An attempt was made to read an object that is not registered in the interface repository. Rerun the SOM compiler to update the interface repository.

38 **SOMPERROR_DUPLICATE_OBJECT_ID**
Attempted to initialize an object with the same ID as another object already initialized.

Replication Framework Error Codes

Given below are the codes returned by Replication Framework methods; the reference manual page on each method states which code a method may return.

Value	Description	Explanation
500	**SOMR_TIMEOUT**	— Possible actions are (1) to retry or (2) to terminate.
501	**SOMR_OK**	
501	**SOMR_GRANTED**	
502	**SOMR_UNAUTHORIZED**	— The likely cause is that the *.scf* file is inaccessible. Or it could be a reader trying to update the replicated object. Recovery action is to ensure proper access.
503	**SOMR_TRYLATER**	— Possible action is to wait for a while and retry the failed operation.
504	**SOMR_DENIED**	— The likely cause is that the *.scf* file is inaccessible. Or it could be a reader trying to update the replicated object. Recovery action is to ensure proper access.
508	**SOMR_MASTER_UNREACHABLE**	— Likely cause is that either the network is down or too slow. Possible actions are (1) Change the time constants through environment variables mentioned earlier. (2) Wait for a while and retry. (3) Ensure that the *.scf* file is accessible.

Messages

It is possible to receive the following messages from the Replication Framework while an application is running. All but the last indicates a misuse of the framework interface or a timing problem.

Replication operation not logged. Probable invalid parameter.
Check the reference manual and rewrite program with appropriate parameters.

somrApplyUpdates in class SOMRReplicbl called. Method must be overridden.
You are using value logging but have not overridden **somrApplyUpdates**.

Warning: Trying to UnPin a replicated object that is not Pinned.
Each call to **somrUnPin** must be preceded by a call to **somrPin**; check your program.

Waiting for Network Transport to be ready...
This message usually appears when the communication buffers are full. When the target application consumes the pending messages, the problem goes away. Occasionally, this can also happen due to a programming error (for example, if a process containing a replica that is the target of update messages blocks indefinitely or enters an infinite loop).

Shutting down listener until some replicas terminate.
This message indicates that the number of replicas reached the permitted maximum.

Listening to connections again.

SOMRERROR Replication Framework Error: N.N.N. Refer to IBM Customer Service.

This message is issued by an internal consistency check in the framework and should never appear. Because of the fault-tolerance of the framework, your application may continue to run correctly. However, the message should be reported so that IBM can provide improvements to the framework.

Environment variable SOMSOCKETS is not defined.

See the topic "Dependence on Sockets DLL" in Chapter 9, "The Replication Framework."

Unable to locate the class <class name> in SOMIR or failed to load the associated dll.

The specified class name is either not found in the implementation repository (indicated by the environment variable SOMIR) or the corresponding dynamic link library could not be found.

Metaclass Framework Error Codes

It is possible to receive the following messages from the Metaclass Framework while an application is running.

60001 An attempt was made to construct a class with **SOMMSingleInstance** as a metaclass constraint. (This may occur indirectly because of the construction of a derived metaclass). The initialization of the class failed because **somInitMIClass** defined by **SOMMSingleInstance** is in conflict with another metaclass that has overridden **somNew**. That is, some other metaclass has already claimed the right to return the value for **somNew**.

60002 An attempt was made to construct a class with **SOMMSingleInstance** as a metaclass constraint. (This may occur indirectly because of the construction of a derived metaclass). The initialization of the class failed because **somInitMIClass** defined by **SOMMSingleInstance** is in conflict with another metaclass that has overridden **somFree**. That is, some other metaclass has already claimed this right to override **somFree**.

60004 An invocation of **somrRepInit** was made with a logging type other than '**o**' or '**v**'.

60005 The **sommBeforeMethod** or the **sommAfterMethod** was invoked on a **SOMRReplicableObject** whose logging type is other than '**o**' or '**v**'. This error cannot occur normally. The likely cause is that some method invoked on another object has overwritten this object's memory.

60006 A Before/After Metaclass must override both **sommBeforeMethod** and **sommAfterMethod**. This message indicates an attempt to create a Before/After Metaclass where only one of the above methods is overridden.

Appendix B. Converting OIDL Files to IDL

This appendix describes how to convert OIDL class descriptions (in .csc files) to IDL class descriptions (in .idl files).

The conversion process involves two steps:

- Converting .csc files to .idl files. This step is largely automatic, and most classes can be converted without intervention.
- Adding extra type information. The difficulty of this step depends largely on how much passthrus are used to define types and constants.

To convert or not to convert

There are several reasons why OIDL users should convert to IDL. Unlike OIDL, IDL offers SOM users multiple inheritance, exception handling, type checking, and automatic descriptor support. In addition, binaries generated from OIDL class descriptions are significantly larger and run more slowly than binaries generated from IDL class descriptions. If users choose *not* to convert their OIDL class descriptions to IDL, however, they can continue to use the SOM Compiler to update their classes, with a few minor changes in protocol. These are:

1. The SOM Compiler no longer generates a .ph (private) and .h (public) file, only a .h file that includes bindings for both private and public methods. To generate a "public" version of the .h file, first generate a .sc file (by invoking the SOM Compiler on the .csc file with the "–ssc" option), then generate a .h file from the .sc file (by invoking the SOM Compiler on the .sc file with the "–sh" option).
2. Because .ph files are no longer used, passthru statements directed toward .ph files should be redirected toward .h files.
3. Passthru statements directed toward .c files should be removed or redirected toward .ih files.
4. Set the environment variables SMADDSTAR=1 and SMNOTC=1:
   For OS/2:
   ```
   SET SMADDSTAR=1
   SET SMNOTC=1
   ```
   For AIX:
   ```
   export SMADDSTAR=1
   export SMNOTC=1
   ```
5. Any methods that return structures should have the modifier "struct" attached to them. For example,
   ```
   UserStruct getUserStruct(), struct;
   ```

Step 1. Converting .csc files to .idl files

The SOM Toolkit supplies a program, **ctoi**, to assist users in converting .csc files to .idl files. Before running **ctoi**, ensure that the directories containing files to convert have all the necessary .sc and .psc files already created. (The SOM Compiler can be run with the –ssc and –spsc options to create .sc and .psc files from a .csc file.)

The conversion process requires a list of all the classes used in the files to be converted, so that forward references to classes can be handled correctly. Store this list of class names in some file (for example, *clsfile*). The name of this file must be specified to the SOM Compiler by the SMCLASSES environment variable:

For OS/2:

```
SET SMCLASSES=clsfile
```

For AIX:

```
export SMCLASSES=clsfile
```

The following command executes the **ctoi** conversion program:

ctoi [file1 file2 ...]

The **ctoi** program generates a .idl file for each specified .csc file.

Once you have run **ctoi**, you should be able to install and run your application program as usual. The following situations, however, may require attention:

- Be sure to change any of your installation batch files or *Makefiles* that explicitly mention .csc, .sc, or .psc files so that they instead refer to .idl files.
- Set the environment variables SMADDSTAR=1 and SMNOTC=1:
  For OS/2:
  ```
  SET SMADDSTAR=1
  SET SMNOTC=1
  ```
  For AIX:
  ```
  export SMADDSTAR=1
  export SMNOTC=1
  ```
- Any methods that return structures should have the modifier “struct” attached to them. For example,
  ```
  UserStruct getUserStruct(), struct;
  ```
- If any of your classes use IDL reserved words as function or variable names, then these names must be changed. Typical cases include “string,” “context”, and “interface.”
- IDL does not permit the following notation for a *struct* type:
  ```
  data:
      struct stat fileStats;
  ```
  Instead, you must add a *typedef* in the IDL *interface* statement that introduces the data element:
  ```
  interface: filemi {
    typedef struct stat stat;
  ...
  #ifdef __SOMIDL__
     implementation {
        stat fileStats;
      ...
     };
  #endif
  };
  ```
  To have the *typedef* emitted into the .h header file, put the *typedef* within the *interface* statement, as shown above. If you don’t want the *typedef* to be emitted in the .h header file, then put it outside the interface statement or in a separate file (to be *#included*). Alternatively, if you *#include* a central header file, then the *typedef* can be put in that header file.

  If you cannot simply add a *typedef*, due to name conflicts in other standard header files, then add a new type (such as *stat_t*, for the example above) and change your .idl files to reflect the new type name.

- The use of unbounded arrays is not allowed in IDL. For example,

  ```
  char *argv[];
  ```

  must be rewritten as:

  ```
  char **argv;
  ```

  or as:

  ```
  #define MAX_SIZE 32
  char *argv[MAX_SIZE];
  ```

- The “unsigned char” type is not supported by IDL. To effectively use unsigned chars, define the type *uchar_t* as follows:

  ```
  typedef octet uchar_t;
  ```

  The SOM Compiler will map this onto an “unsigned char” type in the .h header file.

- IDL does not permit structures to be passed by value. Instead, your methods must pass a pointer to a structure. (Methods can, however, return a structure.)

- Forward references are required in IDL. For all classes not in the ancestry of a class that are used in the *interface* statement for the class, the following statement must precede the class’s *interface* statement:

  ```
  interface <className>;
  ```

- Numeric and string macros that you want to appear in your output files must be mapped onto string constants. For example,

  ```
  #define FILE_NAME_MAX  256
  #define FILE_NAME ”hello.c”
  ```

  must be replaced by:

  ```
  const long FILE_NAME_MAX = 256;
  const string FILE_NAME = ”hello.c”;
  ```

- Public or private instance variables are converted to IDL attributes. However, there are some limitations, as follows: For instance variables that are explicit arrays (such as, `char x[10];` or `short y[20];`) the **ctoi** conversion will result in invalid IDL attributes, because IDL attributes cannot include array declarators. Attributes can be of a type that is an array, such as

  ```
  typedef char myarraytype[10];
  attribute myarraytype myarray;
  ```

  but not an explicit array, as in

  ```
  attribute char myarray[10];        /* not valid */
  ```

  If a .csc file contains a public or private instance variable that is an array, such as

  ```
  char myarray[10];
  ```

  the **ctoi** conversion facility will produce the following in the .idl file it generates:

  ```
  attribute char[10] myarray;
  ```

  This is invalid IDL; it must be fixed manually before the SOM Compiler will accept the .idl file. (It is invalid not only because the array declarator is in the wrong place, but also because attributes cannot include array declarators at all.) To fix it, introduce a typedef that defines an array type, and make that the type of the attribute, as shown:

```
typedef char myarraytype[10];
attribute myarraytype myarray;
```

This limitation does not affect internal instance variables, just public and private ones. (Internal instance variables are not converted to attributes.)

- Most information contained in passthru lines directed to the implementation header (.ih) file should be moved to the implementation (.c) file. In addition, passthru statements directed toward .c files must be removed. (They are no longer allowed.)
- If after running ctoi, you discover that you inadvertently omitted a class name from the file that the SMCLASSES environment variable refers to, it is best to update the class name file, remove the new .idl files, and recreate them using **ctoi**.
- Unlike OIDL, IDL does not include a "private" modifier for data and methods. Instead, private data and methods are surrounded by "#ifdef __PRIVATE__" and "#endif" directives. For example, to declare a method "foo" as a private method within an IDL specification, the following declaration would appear within the interface statement:

```
#ifdef __PRIVATE__
void foo();
#endif
```

To include private data/methods in a compilatiion of a .idl file, the SOM Compiler must be invoked with the "–D__PRIVATE__" option. If any of the data or methods in your .csc files are marked as "private," then when using the SOM Compiler to generate binding files from the .idl files that **ctoi** creates from these .csc files, use the "–D__PRIVATE__" option to have the private data/methods included.

Step 2. Adding type information

IDL, unlike OIDL, is strongly typed. This means that the SOM IDL compiler expects types and constants to be declared before they are referenced. If they are not, the SOM Compiler produces warning messages. Converting from OIDL to IDL does not *require* adding additional typing information (i.e., typedefs and constant definitions), because these warning messages can be safely ignored. If this additional typing information *is* added when converting from OIDL to IDL, however, SOM provides additional functionality not available otherwise. For example, an Interface Repository can be created from a .idl file and the IDL specification can by type–checked only if the file declares types and constants before they are referenced.

In IDL, types (including *typedefs, structs, unions,* and *enums*) are defined in a similar way to C. These types can be emitted into header files if they are defined within the *interface* statement for the class. Type definitions placed outside the *interface* statement are not transferred to header files. See the SOM IDL section of Chapter 4, "SOM IDL and the SOM Compiler," for a complete discussion of defining types and constants in IDL.

Passthrus are not generally needed in IDL to define constants or types, although they may still be used to pass *#include* directives to header files.

Appendix C. SOM IDL Language Grammar

```
specification       : [comment] definition+

definition          : type_dcl  ; [comment]
                    | const_dcl ; [comment]
                    | interface ; [comment]
                    | module    ; [comment]
                    | pragma_stm

module              : module identifier [comment]
                      { [comment] definition+ }

interface           : interface identifier
                    | interface_dcl

interface_dcl       : interface identifier [inheritance] [comment]
                      { [comment] export* } [comment]

inheritance         : : scoped_name {, scoped_name}*

export              : type_dcl ; [comment]
                    | const_dcl ; [comment]
                    | attr_dcl ; [comment]
                    | op_dcl ; [comment]
                    | implementation_body ; [comment]
                    | pragma_stm

scoped_name         : identifier
                    | :: identifer
                    | scoped_name :: identifer

const_dcl           : const const_type identifier = const_expr

const_type          : integer_type
                    | char_type
                    | boolean_type
                    | floating_pt_type
                    | string_type
                    | scoped_name

const_expr          : or_expr

or_expr             : xor_expr
                    | or_expr | xor_expr

xor_expr            : and_expr
                    | xor_expr ^ and_expr

and_expr            : shift_expr
                    | and_expr & shift_expr

shift_expr          : add_expr
                    | shift_expr >> add_expr
                    | shift_expr <<  add_expr
```

```
add_expr            : mult_expr
                    | add_expr + mult_expr
                    | add_expr - mult_expr

mult_expr           : unary_expr
                    | mult_expr * unary_expr
                    | mult_expr / unary_expr
                    | mult_expr % unary_expr

unary_expr          : unary_operator primary_expr
                    | primary_expr

unary_operator      : -
                    | +
                    | ~

primary_expr        : scoped_name
                    | literal
                    | ( const_expr )

literal             : integer_literal
                    | string_literal
                    | character_literal
                    | floating_pt_literal
                    | boolean_literal

type_dcl            : typedef type_declarator
                    | constr_type_spec

type_declarator     : type_spec declarator {, declarator}*

type_spec           : simple_type_spec
                    | constr_type_spec

simple_type_spec    : base_type_spec
                    | template_type_spec
                    | scoped_name

base_type_spec      : floating_pt_type
                    | integer_type
                    | char_type
                    | boolean_type
                    | octet_type
                    | any_type
                    | voidptr_type

template_type_spec  : sequence_type
                    | string_type

constr_type_spec    : struct_type
                    | union_type
                    | enum_type

declarator          : [stars] std_declarator

std_declarator      : simple_declarator
                    | complex_declarator

simple_declarator   : identifier

complex_declarator  : array_declarator
```

```
array_declarator     : simple_declarator fixed_array_size+

fixed_array_size     : [ const_expr ]

floating_pt_type     : float
                     | double

integer_type         : signed_int
                     | unsigned_int

signed_int           : long
                     | short

unsigned_int         : unsigned signed_int

char_type            : char

boolean_type         : boolean

octet_type           : octet

any_type             : any

voidptr_type         : void stars

struct_type          : (struct|exception) identifier
                     | (struct|exception) [comment]
                      { [comment] member* }

member               : type_declarator ; [comment]

union_type           : union identifier
                     | union identifier switch
                       ( switch_type_spec ) [comment]
                       { [comment] case+ }

switch_type_spec     : integer_type
                     | char_type
                     | boolean_type
                     | enum_type
                     | scoped_name

case                 : case_label+ element_spec ; [comment]

case_label           : case const_expr : [comment]
                     | default : [comment]

element_spec         : type_spec declarator

enum_type            : enum identifier { identifier
                       {, identifier}* [comment] }

sequence_type        : sequence < simple_type_spec , const_expr >
                     | sequence < simple_type_spec >

string_type          : string < const_expr >
                     | string

attr_dcl             : [readonly] attribute simple_type_spec
                       declarator {, declarator}*
```

```
op_dcl                : [oneway] op_type_spec [stars] identifier
                         parameter_dcls [raises_expr] [context_expr]

op_type_spec          : simple_type_spec
                      | void

parameter_dcls        : ( param_dcl {, param_dcl}* [comment] )
                      | ( )

param_dcl             : param_attribute simple_type_spec declarator

param_attribute       : in
                      | out
                      | inout

raises_expr           : raises ( scope_name+ )

context_expr          : context ( context_string {, context_string}* )

implementation_body : implementation [comment]
                        { [comment] implementation+ }

implementation        : modifier_stm
                      | pragma_stm
                      | passthru
                      | member

pragma_stm            : #pragma modifier modifier_stm
                      | #pragma somtemittypes on
                      | #pragma somtemittypes off

modifier_stm          : smidentifier : [modifier {, modifier}*] ; [comment]
                      | modifier ; [comment]

modifier              : smidentifier
                      | smidentifier = modifier_value

modifier_value        : smidentifier
                      | string_literal
                      | integer_literal
                      | keyword

passthru              : passthru identifier = string_literal+ ; [comment]

smidentifier          : identifer
                      | _identifier

stars                 | *+
```

Appendix D. Subclassing the Persistence Framework

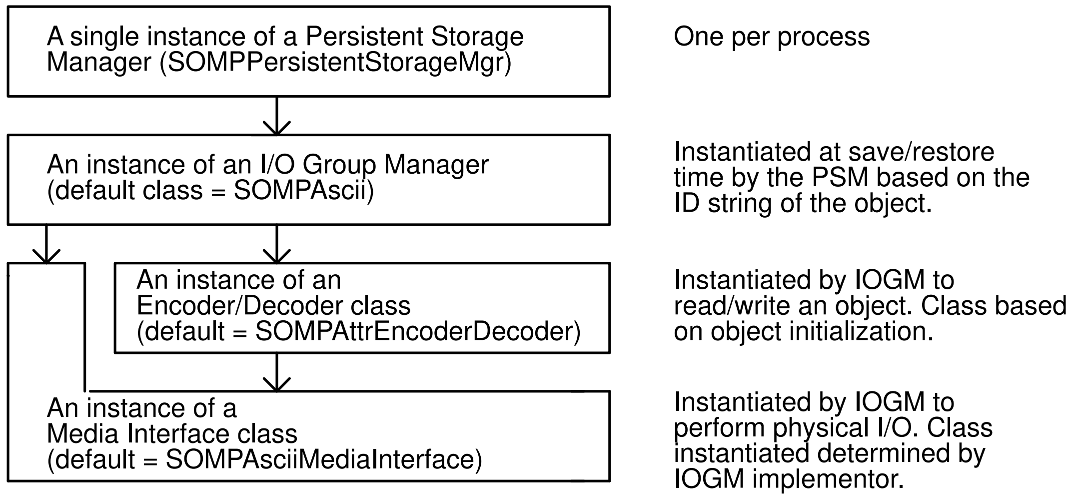
Contents

Appendix D. Subclassing the Persistence Framework

The SOM Persistence Framework will, by default, save and restore SOM objects to a file with a simple format. The format is adequate for many applications; however, some applications may have unique storage requirements for which the default implementation is inadequate. This appendix describes subclassing and modifying the behavior of the SOM Persistence Framework.

Persistence Framework Class Interaction

It is important to understand the various classes that make up the SOM Persistence Framework and how they interact before you attempt to subclass them. Refer to the figure below when reading the following description of the SOM Persistence Framework classes. It is assumed that you have already read Chapter 8, “The Persistence Framework,” earlier in this document and are somewhat acquainted with the Framework.



The primary interface to the SOM Persistence Framework is the Persistent Storage Manager (**SOMPPersistentStorageMgr** class). You invoke methods on this class to save and restore objects. The Persistent Storage Manager responds to requests to save and restore objects by instantiating an object called an I/O Group Manager (a class derived from **SOMPIOGroupMgrAbstract**) to complete the work of actually reading or writing the object. The SOM Persistence Framework supplies a default I/O Group Manager class named **SOMPAscii**.

The format of an individual object is maintained by a class called an Encoder/Decoder which has been derived from **SOMPEncoderDecoderAbstract**. Whenever an instance of an I/O Group Manager is about to read or write an object, it instantiates an Encoder/Decoder. An Encoder/Decoder has a method to read an object and a method to write an object. Every persistent object is associated with at least one encoder/decoder class. The name of the Encoder/Decoder class for a persistent object is stored within the object. By default, the Encoder/Decoder for a persistent object is the class **SOMPAttrEncoderDecoder**. By consulting the SOM Interface Repository, this default class is capable of finding the attributes of a persistent object that have been marked as “persistent,” and storing and restoring them.

Both the I/O Group Manager and the Encoder/Decoder classes share a common third object called a Media Interface (an object derived from **SOMPMediaInterfaceAbstract**). The Media

Interface class provides an interface to the physical media that contains the stored objects. The SOM Persistence Framework supplies an abstract Media Interface class named **SOMPFileMediaAbstract**. This abstract class provides a set of methods to read/write basic IDL type data to and from a file media.

The SOM Persistence Framework supplies two implementations of the **SOMPFileMediaAbstract** class. The first, named **SOMPAsciiMediaInterface**, reads and writes to a file with numeric data written in ASCII. The second, named **SOMPBinaryFileMedia**, is derived from **SOMPAsciiMediaInterface**. It writes numeric data in its binary form to a file.

When the default Encoder/Decoder **SOMPAttrEncoderDecoder** stores/restores an object, it makes use of the write/read methods defined on the **SOMPFileMediaAbstract** class.

Choosing Which Classes to Subclass

You can modify the behavior of the SOM Persistence Framework in a number of ways. You may change the stored format of one of your classes and you may store your objects in a file with a different format than the file produced by the default SOM Persistence Framework. Use the following table as a guide to determine which classes you should be subclassing based on the behavior you would like to change. Consult the sections that follow for more detail on how to do the subclassing.

Behavior	Subclassing
The format of the default file is ok but you would prefer that numbers be written in binary rather than ASCII.	Use the **SOMPBinary** I/O Group Manager. No subclassing is necessary. Initialize your objects with IDs that specify **SOMPBinary**. For example: SOMPBinary:students:0
The format of the default file is ok but you want the elementary pieces of the file to be written in a way that is unique to your application.	Subclass the **SOMPFileMediaAbstract** class and implement a new Media Interface that stores types such as longs, strings, etc., the way you want. Subclass the **SOMPAscii** I/O Group Manager class and override only the **sompInstantiateMediaInterface** method to instantiate your new Media Interface. Initialize your objects with IDs that use the name of your new I/O Group Manager.
Your object contains instance data that the supplied default Encoder/Decoder won't handle. Or, you don't want to make your persistent data attributes since attributes are public.	You should implement a new Encoder/Decoder object for your persistent object.

You want to change the format of the stored object, perhaps because your object data is more conveniently stored in a specialized format.	You should implement a new Encoder/Decoder object for your persistent object.
You want your objects stored in a different data store than the file supplied by default.	This involves subclassing three different SOM Persistence Framework classes. Follow these steps: 1. Decide what sort of data store or storage facility you want to use. Perhaps it is a database or perhaps you want your objects stored in a file that is compatible with some other existing application. 2. Build a new Media Interface class that is derived (either singly or multiply inherited) from **SOMPMediaInterfaceAbstract** or from **SOMPFileMediaAbstract.** Your choice of which base class to derive your new Media Interface from will depend on the data store you choose. **SOMPMediaInterfaceAbstract** is the lowest level class and supports only open and close methods. This base class would be an appropriate choice if your data store reads and writes blocks of aggregated data (for example, a database). If you can read and write in a stream oriented fashion, then the **SOMPFileMediaAbstract** class is a more appropriate base class. Your new Media Interface will be the low level interface to the data store you have chosen. 3. Build a new I/O Group Manager class, derived from **SOMPIOGroupMgrAbstract**, which will route the requests of the Persistent Storage Manager to your new Media Interface. 4. Build new Encoder/Decoders for your objects that are aware of the methods on your new Media Interface. If you have derived your new Media Interface from **SOMPFileMediaAbstract**, it may be possible to use the supplied default Encoder/Decoder **SOMPAttrEncoderDecoder** because it only makes use of the read/write methods defined on **SOMPFileMediaAbstract**.

Implementing New Persistence Framework Classes

The following sections describe the overridable methods for the subclassable classes of the SOM Persistence Framework and how the methods should be implemented to work within the SOM Persistence Framework.

Implementing a new Encoder/Decoder

An Encoder/Decoder is an object that is paired with a persistent object and handles the reading/writing of that persistent object's instance data. An Encoder/Decoder uses a specific class of Media Interface. Before you implement a new Encoder/Decoder, determine which class of Media Interface your Encoder/Decoder will use. The supplied **SOMPAscii** I/O Group Manager passes Encoder/Decoders an instance of the **SOMPAsciiMediaInterface** class. Similarly, the **SOMPBinary** I/O Group Manager passes Encoder/Decoders an instance of the **SOMPBinaryFileMedia**. Become familiar with the methods of your Media Interface class that can be used to save and restore the state of your persistent object.

By implementing a specialized Encoder/Decoder for a persistent object class, you can make assumptions about the methods available on that class or the instance data of that class which the default **SOMPAttrEncoderDecoder** may not. You may also want to store the object in a format other than that of the one provided by **SOMPAttrEncoderDecoder**. There may be many potential Encoder/Decoder's for a persistent object, however, every persistent object is associated with only one Encoder/Decoder at save or restore time.

It is also possible to write general Encoder/Decoders that can be used to read/write many types of objects. The supplied **SOMPAttrEncoderDecoder** is one such general Encoder/Decoder that uses the information available in the SOM Interface Repository.

The class name of the Encoder/Decoder that will be used for a persistent object can be set with the **sompSetEncoderDecoderName** method on the persistent object. This allows you to set Encoder/Decoders on an object-by-object basis. If, however, every object of a class should use the same Encoder/Decoder, it is more appropriate to use the **sompSetClassLevelEncoderDecoderName** method on the class object of your persistent object. This only needs to be done once, however doing it multiple time will do no harm. The **sompSetClassLevelEncoderDecoderName** method can be called at any prior to save/restore time but we suggest you override the somInit method of your persistent object class and do the following (assuming the class name of your new Encoder/Decoder is "mySpecialEncoderDecoder"):

```
ev = SOM_CreateLocalEnvironment();
 _sompSetClassLevelEncoderDecoderName(_somGetClass(somSelf), ev,
                                      "mySpecialEncoderDecoder");
SOM_DestroyLocalEnvironment(ev);
```

Prior to attempting to use a user written Encoder/Decoder, the client program should ensure that the class of the Encoder/Decoder exists by executing its NewClass procedure. For example:

```
mySpecialEncoderDecoderNewClass(0,0);
```

This should also be done in the somInit method of your persistent object. It only needs to be done once, however doing it multiple times will do no harm. If your new Encoder/Decoder class is built into a dynamically loadable class library (see "Creating a SOM Class Library" in Chapter 5, "Implementing Classes in SOM") then the explicit call to the Encoder/Decoder's NewClass procedure in somInit is not necessary.

When an object is stored/restored, the I/O Group Manager uses the Encoder/Decoder returned by the **sompGetEncoderDecoder** method on the persistent object. The

sompGetEncoderDecoder method attempts to find and instantiate an instance of the encoder/decoder class name specified with either **sompSetEncoderDecoderName** or **sompSetClassLevelEncoderDecoderName**. By default, **sompGetEncoderDecoder** returns an instance of class **SOMPAttrEncoderDecoder**.

Chapter 8 contains a complete example of a user–written Encoder/Decoder. The toolkit samples contains the complete implementation of the example.

Encoder/Decoder methods

The following methods of **SOMPEncoderDecoderAbstract** must be overridden to build your own Encoder/Decoder.

sompEDWrite — This method is invoked by an I/O Group Manager when the I/O Group Manager determines it is time to store the instance data of a persistent object. The I/O Group Manager passes this method the object to store and a prepared MediaInterface (i.e. instantiated and opened).

The implementor of the Media Interface passed to your Encoder/Decoder will have provided an interface to store the data of your object. Consult the interface definition of the Media Interface.

If you're using either of the supplied Media Interfaces derived from **SOMPFileMediaAbstract**, the Media Interface passed to **sompEDWrite** will provide a set of methods that can be used in a sequential fashion to write basic standard IDL types to a physical media. The **sompEDWrite** method should make use of these methods whenever possible to improve the portability of the Encoder/Decoder. For example, if the object to be stored has as its persistent data a character string, the implementation of **sompEDWrite** should at some point make the following method call:

```
_sompWriteString(mediaInterface, ev, stringData);
```

By making use of the Media Interface methods to write basic standard IDL types, a single Encoder/Decoder implementation can be used for potentially many Media Interface's.

To provide a given Media Interface implementation an opportunity to collect all the individual write requests made during **sompEDWrite** into one physical I/O operation, it is suggested that your implementation of the **sompEDWrite** method only make use of the Media Interface's "write" methods and no others.

Note: The write operations made to a Media Interface by the **sompEDWrite** method should be a mirror image of the read operations made by the **sompEDRead** method.

sompEDRead — This method is invoked by an I/O Group Manager when the I/O Group Manager determines it is time to restore the instance data of a persistent object. The I/O Group Manager passes this method the object to restore and a prepared

MediaInterface (i.e. instantiated and opened). The implementor of the Media Interface passed to your Encoder/Decoder will have provided an interface to restore the data of your object. Consult the interface definition of the Media Interface.

If you're using either of the supplied Media Interfaces derived from **SOMPFileMediaAbstract**, the Media Interface passed to **sompEDRead** will provide a set of methods that can be used in a sequential fashion to read basic standard IDL types from a physical media. The **sompEDRead** method should make use of these methods whenever possible to improve the portability of the Encoder/Decoder. For example, if the object to be restored has as its persistent data a character string, the implementation of **sompEDRead** should at some point make the following method call:

```
_sompReadString(mediaInterface, ev, &stringData);
```

By making use of the Media Interface methods to read basic standard IDL types, a single Encoder/Decoder implementation can be used for potentially many Media Interface's.

To provide a given Media Interface implementation an opportunity to collect all the individual read requests made during **sompEDRead** into one physical I/O operation, it is suggested that your implementation of the **sompEDRead** method only make use of the Media Interface's "read" methods and no others.

Note: The read operations made to a Media Interface by the **sompEDRead** method should be a mirror image of the read operations made by the **sompEDWrite** method.

Implementing a new or enhanced Media Interface

A Media Interface is the low–level interface to some physical media. An instance of a Media Interface is the means by which the I/O Group Manager and Encoder/Decoder classes save/restore an object's state. There are two abstract Media Interface classes supplied in the SOM Persistence Framework: **SOMPMediaInterfaceAbstract** is the base class and **SOMPFileMediaAbstract** is derived from it, as follows:

- **SOMPMediaInterfaceAbstract** defines only the **sompOpen** and **sompClose** methods. It is expected that at the very least, a media will require you to open and close it.
- The **SOMPFileMediaAbstract** defines a set of methods, in addition to **sompOpen** and **sompClose**, to read/write to a file and to move the file pointer. There are also a number of methods to read/write basic standard IDL defined data types.

Before implementing a new Media Interface, decide whether:

1. You can simply enhance an existing Media Interface, or
2. You must build a new Media Interface

A Media Interface is used by both Encoder/Decoders and I/O Group Managers so your design decisions regarding the Media Interface will affect these other classes. Enhancing an existing Media Interface usually has minimal impact on existing Encoder/Decoders and I/O Group

Managers. Implementing a new Media Interface with new methods for reading/writing object data usually requires a completely new I/O Group Manager and new Encoder/Decoders that know about the new methods.

Enhancing an existing Media Interface

You may want to enhance one of the supplied File Media Interface classes, for example, to provide new methods that read/write some application specific data type. You can use these new methods in new Encoder/Decoders for your objects that contain the specific data type. To enhance a supplied file Media Interface do the following:

1. Create a new class that is derived from either **SOMPAsciiMediaInterface** or **SOMPBinaryFileMedia**. For example, a class named EnhancedMediaInterface, derived from **SOMPAsciiMediaInterface**, here adds two new methods to read and write the new *myAppType* data type:

```
#include <somp.idl>
#include <fmi.idl>

interface EnhancedMediaInterface : SOMPAsciiMediaInterface

// This is a sample SOMPAsciiMediaInterface which has the ability to
// read and write my application data type.
   struct myAppType {
     myAppType *next;
     myAppType *prev;
   };
void myReadAppType(inout myAppType list);
// Reads data of my application's type.

void myWriteAppType(in myAppType list);
// Writes data of my application's type.

#ifdef __SOMIDL__
implementation
{
  callstyle=idl;
  dllname="emi.dll";
  releaseorder: myReadAppType, myWriteAppType;
};
#endif /* __SOMIDL__ */

};
```

   The implementation of this class is included in the Toolkit samples.

2. Create a new I/O Group Manager class which is derived from the I/O Group Manager class that used the Media Interface you are enhancing. In this case, that is the **SOMPAscii** class. I/O Group Manager objects instantiate the Media Interface used to read/write an object so, therefore, in order to use your new Media Interface you must create a new I/O Group Manager.

3. In the interface definition of your I/O Group Manager, override the **somInit** and **sompInstantiateMediaInterface** methods. For example, here we have defined a new I/O Group Manager named EnhancedAscii, derived from the supplied **SOMPAscii**.

```
#include <fsagm.idl>
#include <somp.idl>

interface EnhancedAscii : SOMPAscii
// This IO Group Manager class uses the sample
// EnhancedMediaInterface class.
{

#ifdef __SOMIDL__

implementation
{
   callstyle=idl;
   dllname="emi.dll";

// Method Modifiers
   sompInstantiateMediaInterface: override;
   somInit: override;

};
#endif /* __SOMIDL__ */

};
```

The implementation of this class is included in the Toolkit samples.

4. Write new Encoder/Decoders to make use of the new methods that you have defined on your enhanced Media Interface. In the **sompEDWrite** method you would make use of your new myWriteAppType method. In the **sompEDRead** method you would make use of myReadAppType.

5. Build your new classes into a dynamically loadable library. If your new classes are not in a dynamically loaded library, the Persistent Storage Manager will not be able to find and instantiate your new I/O Group Manager class. This subject is covered in chapter 4.

   Note: An alternative to building a dynamically loadable library would be to explicitly call the EnhancedAsciiNewClass and EnhancedMediaInterface procedure prior to any attempt to save/restore.

6. To use the new classes when your objects are saved/restored, initialize your persistent objects with IDs that specify the class name of the new I/O Group Manager. An ID string for the examples shown above might be:

   ```
   EnhancedAscii:employee:0
   ```

Creating a new Media Interface

If the file interface provided by the supplied **SOMPAsciiMediaInterface** class is inadequate or inappropriate for your application, you can create a new Media Interface. To accomplish this you must:

1. Create a new class that is derived from **SOMPMediaInterfaceAbstract**. You may have already built a class that provides access to the media of your choice. To use that class within the SOM Persistence Framework, use multiple inheritance to derive your new class from your existing class and **SOMPMediaInterfaceAbstract**. For example, suppose you have a class that implements a cursor for a database. This class could be used as a Media Interface. You could define a new class that works with the SOM Persistence Framework as follows. The new class has the methods defined for both classes.

```
interface SqlMedia : SOMPMediaInterfaceAbstract, myDatabaseCursor
{
    /* ... */
   somInit: override;
   somUninit: override;
}
```

If you prefer to encapsulate your existing class, you could define your new class as follows:

```
interface SqlMedia : SOMPMediaInterfaceAbstract
{
   ...
   // Instance data
   myDatabaseCursor dbc;
   ...
}
```

2. Create a new I/O Group Manager class to use your new Media Interface. It is unlikely you will be able to use any of the supplied implementations of an I/O Group Manager as a base, so start with the template I/O Group Manager described in “Template for an I/O Group Manager” later in this appendix.
3. Write new Encoder/Decoders to make use of your new Media Interface.
4. To use the new classes when your objects are saved/restored, initialize your persistent objects with IDs that specify the class name of the new I/O Group Manager. Based on the example above you might use:

   ```
   SqlMedia:employee:0
   ```

5. Build all your new classes into a dynamically loadable library. If your new classes are in a dynamically loaded library, it will be possible for SOM to find and instantiate them. Alternatively, you can explicitly call the <*className*>**NewClass** procedure of each of your classes during the initialization of your program. For example:

   ```
   SqlMediaNewClass(0,0);
     ...
   ```

Media Interface methods

This section lists the Media Interface methods for the **SOMPFileMediaAbstract** class. Most of these methods should be overridden in a new File Media Interface, however those that should not be are noted. To reuse existing code, we recommend that you subclass from either **SOMPAsciiMediaInterface** or **SOMPBinaryFileMedia**.

sompOpen — This method is for opening the media to which your new class will be reading and writing. This method is typically invoked soon after the object is instantiated. Note that this method, which is inherited from the base class **SOMPMediaInterfaceAbstract**, has no parameters. Class implementations of the base class should introduce initialization methods (such as **sompInitReadWrite** and **sompInitReadOnly**) to provide any information required to accomplish the open.

Typically, you would not have to override this method.

sompClose — This method is for closing the media. This method is typically invoked when the Persistent Storage Manager instructs its I/O Group Manager to free its Media Interface. You should only override this method if you have overridden **sompOpen**.

sompInitReadWrite — Initializes the Media Interface for reading and writing the given file name. Override this method only if **sompOpen** has been overridden.

sompInitReadOnly — Initializes the Media Interface for reading only the given file name. Override this method only if **sompOpen** has been overridden.

sompInitSpecific — This method allows for other forms of initialization not handled by either **sompInitReadWrite** or **sompInitReadOnly**. Override this method only if **sompOpen** has been overridden.

sompSeekPosition — Set the file pointer to an offset relative to the beginning of the file. Override this method only if **sompOpen** has been overridden.

sompSeekPositionRel — Set the file pointer to an offset relative to the current file pointer. Override this method only if **sompOpen** has been overridden.

sompGetOffset — Return the current file pointer offset. Override this method only if **sompOpen** has been overridden.

sompReadBytes — Read a block of bytes. Override this method only if **sompOpen** has been overridden.

sompWriteBytes — Write a block of bytes. Override this method only if **sompOpen** has been overridden.

sompWriteOctet — Write the standard IDL defined octet to a file. This method makes use of **sompWriteBytes** and therefore you will probably not have to override this method.

sompWriteShort — Write the standard IDL defined short to a file. This method makes use of **sompWriteBytes** and therefore you will probably not have to override this method.

sompWriteUnsignedShort — Write the standard IDL defined unsigned short to a file. This method makes use of **sompWriteBytes** and therefore you will probably not have to override this method.

sompWriteLong — Write the standard IDL defined long to a file. This method makes use of **sompWriteBytes** and therefore you will probably not have to override this method.

sompWriteUnsignedLong — Write the standard IDL defined unsigned long to a file. This method makes use of **sompWriteBytes** and therefore you will probably not have to override this method.

sompWriteDouble — Write the standard IDL defined double to a file. This method makes use of **sompWriteBytes** and therefore you will probably not have to override this method.

sompWriteFloat — Write the standard IDL defined float to a file. This method makes use of **sompWriteBytes** and therefore you will probably not have to override this method.

sompWriteCharacter — Write a character to a file. This method does *not* make use of **sompWriteBytes**. You *should* override this method and implement it to write a character to your media.

sompWriteSomobject — This method is provided for Encoder/Decoders to write out contained objects. For example, if object A refers to object B, then when object A is stored its Encoder/Decoder would call **sompWriteSomobject** to store the contained object B. This method makes use of **sompWriteCharacter** and **sompWriteString**. Provided that these methods are implemented in your new Media Interface, you should not override this method.

sompWriteString — Write a null terminated string to a file. This method makes use of **sompWriteBytes** and therefore you will probably not have to override this method.

sompWriteLine — Writes the newline terminated string to a file. This method does *not* make use of **sompWriteBytes**. You should override and implement this method. This method is equivalent to the fputs() c library function. The terminating null character (\0) is not written. This method does NOT append a newline character (\n) to the given string before writing. If the user of this method intends to restore the string written via **sompWriteLine** with **sompReadLine**, the user must put the newline character in the string before calling this method.

sompReadOctet — Read the standard IDL defined octet from a file. This method makes use of **sompReadBytes** and therefore you will probably not have to override this method.

sompReadShort — Read the standard IDL defined short from a file. This method makes use of **sompReadBytes** and therefore you will probably not have to override this method.

sompReadUnsignedShort — Read the standard IDL defined unsigned short from a file. This method makes use of **sompReadBytes** and therefore you will probably not have to override this method.

sompReadLong — Read the standard IDL defined long from a file. This method makes use of **sompReadBytes** and therefore you will probably not have to override this method.

sompReadUnsignedLong — Read the standard IDL defined unsigned long from a file. This method makes use of **sompReadBytes** and therefore you will probably not have to override this method.

sompReadDouble — Read the standard IDL defined double from a file. This method makes use of **sompReadBytes** and therefore you will probably not have to override this method.

sompReadFloat — Read the standard IDL defined float from a file. This method makes use of **sompReadBytes** and therefore you will probably not have to override this method.

sompReadCharacter — Read a character from a file. This method does *not* make use of **sompReadBytes**. You *should* override this method and implement it to write a character to your media.

sompReadSomobject — This method is provided for Encoder/Decoders to read in contained objects that were written using **sompWriteSomobject**. For example, if object A refers to object B, then when object A is restored its Encoder/Decoder would call **sompReadSomobject** to restore the contained object B. This method makes use of **sompReadCharacter** and **sompReadString**. Provided that these methods are implemented in your new Media Interface, you should not override this method.

sompReadString — Read and return a null terminated string. This method allocates block of storage large enough to contain the string. You must, at some point, free the string returned by this method with SOMFree(). This method makes use of **sompReadLong** and **sompReadBytes**. Provided that these methods are implemented in your new Media Interface, you should not override this method.

sompReadStringToBuffer — Read and return a null terminated string in the preallocated buffer provided by the caller. This method makes use of **sompReadLong** and **sompReadBytes**. Provided that these methods are implemented in your new Media Interface, you should not override this method.

sompReadLine — Read a string up to and including the first newline character (\n) into the preallocated buffer provided by the caller. Use this method for reading strings stored with **sompWriteLine**. This method does *not* make use of **sompReadBytes**. You should override and implement this method.
Note: If the string read is larger than size of the buffer given it is truncated to fit in the given buffer. A null character (\0) is appended. The newline character, if read, is included in the string.

sompWriteTypeCode — Do not override this method. Instead, override methods **sompWriteLong** and **sompWriteBytes**, the methods used by **sompWriteTypeCode**.

sompReadTypeCode — Do not override this method. Instead, override methods **sompReadLong** and **sompReadBytes**, the methods used by **sompReadTypeCode**.

Implementing a new I/O Group Manager

This section describes the I/O Group Manager methods that should be overridden to implement a new I/O Group Manager. It also describes the behavior of the supplied I/O Group Manager **SOMPAscii**. You may or may not wish to emulate that behavior in your new I/O Group Manager. Methods that must be implemented in response to requests from the Persistent Storage Manager are noted. You may also want to refer to “Persistent Storage Manager Interaction” in the next section to become acquainted with the order in which the Persistent Storage Manager invokes methods on an I/O Group Manager.

How you implement an I/O Group Manager class may have implications for the Encoder/Decoder and Media Interface classes you use. You may need to re-implement these other classes.

Refer to the template for an I/O Group Manager that is shipped with sample in the SOMobjects/2 Toolkit when reading the following section. See “Template for an I/O Group Manager” later in this appendix.

When implementing a new I/O Group Manager:

1. You must build your new I/O Group Manager class into a dynamically loadable library or, explicitly call the NewClass procedure of your new I/O Group Manager class during the initialization of your application.

2. If your class is built into a dynamically loadable library, you must specify the name of the loadable library file in the interface definition (in your .idl file) for your new I/O Group Manager. For example:

```
#ifdef __SOMIDL__
implementation
{
     callstyle=idl;
     dllname="temp.dll";  /* load from here */
     ...
};
#endif /* __SOMIDL__ */
```

3. For somFindClass to find your new I/O Group Manager class, you must include the –u flag when you run the SOM compiler on your new I/O Group Manager .idl file. This updates the SOM Interface Repository so that SOM can find and load your I/O Group Manager class.

I/O Group Manager methods

sompNewMediaInterface — This method is invoked by the Persistent Storage Manager to prepare it for storing or restoring a group of objects. It must be implemented in a new I/O Group Manager. This method should invoke **sompInstantiateMediaInterface** whenever it needs a new instance of a Media Interface.

The **SOMPAscii** I/O Group Manager invokes the **sompInstantiateMediaInterface** on itself to instantiate a Media Interface object. It uses the Media Interface object to store information about the objects in the group. Once the Media Interface has been instantiated, **SOMPAscii** invokes the **sompInitReadWrite** and **sompOpen** methods on its Media Interface.

sompInstantiateMediaInterface—This method is provided in the interface to return an instance of a Media Interface class that the I/O Group Manager may use. It should be used when the method **sompNewMediaInterface** is invoked on an I/O Group Manager. The Persistent Storage Manager does not make use of this method directly.

This method is provided primarily so that others may override it and easily replace the media interface your I/O Group Manager uses.

The **SOMPAscii** I/O Group Manager instantiates a Media Interface of class **SOMPAsciiMediaInterface**.

sompGetMediaInterface — This method is provided in the interface to return the Media Interface that was instantiated as a result of **sompNewMediaInterface**. The Persistent Storage Manager does not make use of this method directly. This method provides a way for subclasses of your I/O Group Manager to get at the Media Interface you have instantiated.

sompFreeMediaInterface — This method is invoked by the Persistent Storage Manager to tell the I/O Group Manager that it no longer needs access to the Media Interface it prepared when **sompNewMediaInterface** was invoked. The method must be implemented in a new I/O Group Manager.

SOMPAscii invokes the **sompClose** method on its Media Interface and then frees the Media Interface.

sompWriteGroup — This method is invoked by the Persistent Storage Manager in response to **sompStoreObject** to write at least the given object, or at most, the given object and all the others grouped with it. The method must be implemented in a new I/O Group Manager. Refer to “Template for an I/O Group Manager” later in this appendix for a sample of how this method should be implemented.

Depending on the data store of objects your I/O Group Manager is managing, you may want to write just the object given or all the objects grouped with the object passed to this method. To get all the objects grouped with the given object, invoke the **sompGetIOGroup** method on the object. **sompGetIOGroup** returns a **SOMPIOGroup** object that you can iterate over to store each of the objects. If your implementation intends to store just the object given, **sompWriteGroup** should return FALSE. If it attempts to store all of the objects grouped with the given object, then **sompWriteGroup** should return TRUE.

SOMPAscii iterates through the collection returned by **sompGetIOGroup**, stores each object via the object’s Encoder/Decoder, and returns TRUE.

sompGroupExists — This method is invoked by the Persistent Storage Manager to determine if a group exists. The method returns either TRUE or FALSE and must be implemented in a new I/O Group Manager.

SOMPAscii simply determines if the given file exists.

sompObjectInGroup — This method is invoked by the Persistent Storage Manager to determine if an object exists within the given group. The method returns either TRUE or FALSE, depending on whether the object in the group of the given ID exists, and must be implemented in a new I/O Group Manager.

SOMPAscii searches to see if an object with the given ID exists.

sompMediaFormatOk — This method is provided in the interface to be a checkpoint where the I/O Group Manager can determine if it is dealing with an understandable file. Implementations of this method would verify a magic number was correct or a version number was correct, for example. The method is not invoked by the Persistent Storage Manager.

sompDeleteObjectFromGroup—This method is invoked by the Persistent Storage Manager to delete an object from a group. The method must be implemented in a new I/O Group Manager.

SOMPAscii searches to see if an object with the given ID exists and if so, it is deleted.

sompReadGroup — This method is invoked by the Persistent Storage Manager to return the persistent object specified by the ID passed to the method. The method must be implemented in a new I/O Group Manager.

The **sompReadGroup** method must at the very least, instantiate and initialize (via **sompInitGivenId**) the object represented by the ID it has been passed. It is up to the I/O Group Manager implementor to determine if the data of the instantiated object or objects should be read at this point. If the data is not read, the method should set the state of the object(s) to *unstable* with

```
_sompSetState(thisPo, ev, SOMP_STATE_UNSTABLE);
```

The data of an object need not be read at this point. The **sompReadObjectData** method exists for reading an object's data and will be called by the Persistent Storage Manager when required . If object data is read by the **sompReadGroup** method then the object must have its state set to SOMP_STATE_STABLE and the **sompReadObjectData** method should be able to sense that the object's data has been read and not attempt to read the data a second time. Your implementation may require that you instantiate all the objects grouped with the object specified by the given ID. At most, the **sompReadGroup** method should instantiate all the objects in the group, read all their data and mark the object's as SOMP_STATE_STABLE.

Be aware that if you implement **sompReadGroup** to read in an object's data, the Persistent Storage Manager method **sompRestoreObjectWithoutChildren** will not behave as described in chapter 8. Instead, child objects will be completely restored to *stable* objects.

The supplied **SOMPAscii** I/O Group Manager implements this method by instantiating and initializing all the objects stored in the group. It reads no object data and leaves all objects as SOMP_STATE_UNSTABLE. The template I/O Group Manager in "Template for an I/O Group Manager" later in this appendix shows one possible implementation of the **sompReadGroup** method. In the implementation, the very least that must be done by this method is done. Note that the class name of the object represented by the given ID must be restored from the data store where the object is stored. Without the class name, it is impossible to instantiate an object of the stored class.

sompReadObjectData — This method is invoked by the Persistent Storage Manager to set the given object's persistent data to the state it was when the object was stored. The method must be implemented in a new I/O Group Manager. The Persistent Storage Manager only calls this method if the state of the object returned by **sompReadGroup** is SOMP_STATE_UNSTABLE. How this method is implemented depends on how **sompReadGroup** has been implemented . Note the example implementation in "Template for an I/O Group Manager".

Persistent Storage Manager interaction

The following sections describe the methods that are called and the order in which they are called when objects are stored and restored by the Persistent Storage Manager. It is important for anyone who implements a new I/O Group Manager class to understand how the Persistent Storage Manager will be making use of the their I/O Group Manager. The list of methods below is not complete, however, it is sufficient to implement a new I/O Group Manager.

*Storing objects with **sompStoreObject***

The following methods are invoked to complete the storage of an object when the **sompStoreObject** method is invoked on the Persistent Storage Manager.

1. **sompGetPersistentId** on Persistent Object

   From the ID of the object, the Persistent Storage Manager determines the class name of the I/O Group Manager class that will be used to complete the storage of the object. Therefore, if you implement a new I/O Group Manager, you must modify your source code so that the IDs of your objects contain the class name of your new I/O Group Manager. If you are using system–assigned IDs you must implement a new IdAssigner class.

2. **sompGetIOGroupMgrClassName** on the object's ID

   At this point the Persistent Storage Manager instantiates a new I/O Group Manager object based on the class name in the object's ID.

3. **sompGetIOGroupName** on the object's ID.

   The name of the object's group is returned from the object's ID. The returned name is passed to the I/O Group Manager on the **sompNewMediaInterface** method. The group-name portion of an object's persistent ID that is returned by **sompGetIOGroupName** is unique to the implementation of a given I/O Group Manager. For the supplied **SOMPAscii** I/O Group Manager, the group name corresponds to the file in which the object(s) will be stored. However, if you implemented a group manager that stored objects in a database, the group name portion of the object ID might contain the name of a database and the table within the database in which the object is stored. For example, your persistent ID string might look like:

   ```
   myDBGM:mydatabase,employee:0
   ```

4. **sompNewMediaInterface** on I/O Group Manager

   The Persistent Storage Manager tells the I/O Group Manager to prepare the media interface the I/O Group Manager will use to store the object(s). For the supplied **SOMPAscii** class, this amounts to telling it the name of the file it should open.

5. **sompWriteGroup** on I/O Group Manager

   The Persistent Storage Manager passes the object requested to be stored to the I/O Group Manager. The I/O Group Manager responds by storing the specified object and possibly all objects grouped with the specified object. The choice is up to the I/O Group Manager implementor.

6. **sompFreeMediaInterface** on I/O Group Manager

   The Persistent Storage Manager tells the I/O Group Manager that a media interface is no longer required.

7. The Persistent Storage Manager now frees the I/O Group Manager object.

   The object has been written and the instance of the I/O Group Manager is freed.

I/O Group Manager implementors should be aware that as child objects of the initially stored object are stored, new instances of their I/O Group Manager class will be created.

*Restoring objects with **sompRestoreObject***

The following methods are invoked to complete the restoration of an object when **sompRestoreObject** is invoked on the Persistent Storage Manager.

1. **sompGetIOGroupMgrClassName** on the given persistent object ID

   The Persistent Storage Manager instantiates a new I/O Group Manager object based on the class name in the object's ID.

2. **sompGetIOGroupName** on the given persistent object ID

   The group name returned is used to see if the group exists.

3. **sompGroupExists** on I/O Group Manager

   Before attempting to restore an object from a group, a check is made to see if a group even exists. If the group does not exist, the Persistent Storage Manager returns at this point after freeing the I/O Group Manager.

4. **sompNewMediaInterface** on I/O Group Manager

   The Persistent Storage Manager tells the I/O Group Manager to prepare the media interface the I/O Group Manager will use to restore the object(s). The group name returned by **sompGetIOGroupName** is passed to **sompNewMediaInterface**.

5. **sompObjectInGroup** on I/O Group Manager

   Before attempting to restore an object from a group, the Persistent Storage Manager checks to see if the object exists in the group. If the object is not in the group, the Persistent Storage Manager returns at this point after freeing the I/O Group Manager.

6. **sompReadGroup** on I/O Group Manager

   The **sompReadGroup** method returns, at a minimum, the persistent object corresponding to the persistent object ID originally passed to the **sompRestoreObject** method. The object may or may not have been fully restored yet. That is, its instance data may not yet have been read from storage and set. The Persistent Storage Manager detects if the object has been fully restored by checking its state as follows:

   ```
   _sompCheckState(restoredObject, ev, SOMP_STATE_UNSTABLE)
   ```

   If the check returns TRUE, further processing is done later by the Persistent Storage Manager.

7. **sompFreeMediaInterface** on I/O Group Manager

   The Persistent Storage Manager tells the I/O Group Manager that a media interface is no longer required.

8. The Persistent Storage Manager now frees the I/O Group Manager object.

9. **sompReadObjectData** on I/O Group Manager

   If further processing is required to fully restore the requested object, the Persistent Storage Manager once again goes through the steps of instantiating a new I/O Group Manager and invoking the **sompNewMediaInterface** method on it. The Persistent Storage Manager then invokes the **sompReadObjectData** method on the I/O Group Manager passing it the persistent object whose data is to be read.

10. **sompFreeMediaInterface** on I/O Group Manager

    The Persistent Storage Manager tells the I/O Group Manager that a media interface is no longer required.

11. The Persistent Storage Manager now frees the I/O Group Manager object.

12. **sompActivated** on the restored persistent object

    Once an object is fully restored, via either **sompReadGroup** or **sompReadObjectData**, the **sompActivated** method is invoked on it to allow the object to perform any initialization related to restoration prior to being returned to the caller of **sompRestoreObject**. By default, the **sompActivated** method simply sets the state of the object as being fully restored.

Template for an I/O Group Manager

The following class definition and implementation can be used as a starting point for building your own I/O Group Manager. You may also want to refer to “An Example I/O Group Manager and Media Interface Implementation” in the following section for an example I/O Group Manager and Media Interface that work together to store objects into an OS/2 .INI file.

```
/*
 *  @(#)template.idl 1.9 5/4/93 11:01:32 [5/4/93] (c)IBM Corp. 1993
 */
#ifndef template_idl
#define template_idl

#include <iogma.idl>
#include <somp.idl>
#include <somperrd.idl>
#include <sompstad.idl>

interface SOMPTemplate : SOMPIOGroupMgrAbstract

// This is a template IO Group Manager class.
//
// This class is a starting point for someone who intends to build a
// new IO Group Manager to store/restore objects to the container of
// their choice.

{#ifdef __SOMIDL__

implementation
{
   callstyle=idl;

   dllname=”template.dll”;

// Class Modifiers
   filestem = template;

// Internal Instance Variables
   SOMPMediaInterfaceAbstract mia;
```

```
// Method Modifiers
   sompNewMediaInterface: override;
   sompGetMediaInterface: override;
   sompFreeMediaInterface: override;
   sompInstantiateMediaInterface: override;
   sompWriteGroup: override;
   sompReadGroup: override;
   sompReadObjectData: override;
   sompDeleteObjectFromGroup: override;
   sompGroupExists: override;
   sompObjectInGroup: override;
   sompMediaFormatOk: override;
   somInit: override;
   somUninit: override;

};
#endif /* __SOMIDL__ */

};

#endif  /* template_idl */
```

The SOMPTemplate implementation

Refer to the preceding section entitled "I/O Group Manager methods" when reading through this code.

```
#include <somp.h>
#include <stdlib.h>
#include <stdio.h>
#include <iogrp.h>

#define SOMPTemplate_Class_Source
#include <template.ih>

SOM_Scope void  SOMLINK sompNewMediaInterface(SOMPTemplate somSelf,
                                    Environment *ev, string IOInfo)
{
    SOMPTemplateData *somThis = SOMPTemplateGetData(somSelf);
    SOMPTemplateMethodDebug("SOMPTemplate","sompNewMediaInterface");

    if (_mia == NULL) {
       _mia = _sompInstantiateMediaInterface(somSelf, ev);
       _sompInitReadWrite(_mia, ev, IOInfo);
       if (ev->_major == NO_EXCEPTION) {
          _sompOpen(_mia, ev);
       } /* endif */
    } /* endif */
}

SOM_Scope SOMPMediaInterfaceAbstract  SOMLINK
sompGetMediaInterface(SOMPTemplate somSelf, Environment *ev)
{
    SOMPTemplateData *somThis = SOMPTemplateGetData(somSelf);
    SOMPTemplateMethodDebug("SOMPTemplate","sompGetMediaInterface");
    return (_mia);
}
```

```
SOM_Scope void  SOMLINK sompFreeMediaInterface(SOMPTemplate somSelf,
Environment *ev)
{
    SOMPTemplateData *somThis = SOMPTemplateGetData(somSelf);
    SOMPTemplateMethodDebug("SOMPTemplate",
                         "sompFreeMediaInterface");
    if (_mia) {
       _sompClose(_mia, ev);
       _somFree(_mia);
       _mia = NULL;
    } /* endif */
}

SOM_Scope SOMPMediaInterfaceAbstract  SOMLINK
sompInstantiateMediaInterface(SOMPTemplate somSelf, Environment *ev)
{
    SOMPMediaInterfaceAbstract mia = NULL;
    SOMPTemplateData *somThis = SOMPTemplateGetData(somSelf);
    SOMPTemplateMethodDebug("SOMPTemplate",
                             "sompInstantiateMediaInterface");

    /* mia = <mediaInterfaceClassName>New(); */
    mia = SOMPAsciiMediaInterfaceNew(); /* sample only */
    return(mia);
}

SOM_Scope boolean SOMLINK sompWriteGroup(SOMPTemplate somSelf,
                                    Environment *ev,
                                    SOMPPersistentObject storeObj)
{
    SOMPIteratorHandle hit;
    SOMObject thisPo;
    SOMPIOGroup thisGroup;
    SOMPEncoderDecoderAbstract ed;

    SOMPTemplateData *somThis = SOMPTemplateGetData(somSelf);
    SOMPTemplateMethodDebug("SOMPTemplate","sompWriteGroup");

#ifdef TMP_WRITE_ALL
/* Write all objects grouped with the given object */
    thisGroup = _sompGetIOGroup(storeObj, ev);
    hit = _sompNewIterator(thisGroup, ev);

    /* Get each of the objects from the group. If it should be
       stored, then instantiate an Encoder/Decoder object and
       invoke its sompEDWrite method.                          */
    while ( ((thisPo = _sompNextObjectInGroup(thisGroup, ev,
              hit))!=NULL) &&
              ev->_major == NO_EXCEPTION) {
```

```
      if (_sompIsDirty(thisPo, ev)) {
         _sompPassivate(thisPo, ev); /* Tell object to prepare to be
                                        stored. */
         /* ...
            At this point, the IO Group Mgr must
            store anything about the object necessary in order to
            find it again in the group container. You may want to
            store any of:
             - how many objects are in the group container
             - where the object is located
             - mapping between ID and the class name of the object
               (used when the object is restored).
             - the class name of the encoder/decoder used by the
               object
             - etc.
            ... */
         ed = _sompGetEncoderDecoder(thisPo, ev);
         /* Now, write the object data */
         if (ed) {
            _sompEDWrite(ed, ev, _sompGetMediaInterface(somSelf,
                                  ev), thisPo);
            _sompFreeEncoderDecoder(thisPo, ev);
            _sompClearState(thisPo, ev, SOMP_STATE_DIRTY);
            /* ...
               any other unique processing
               ... */
         } else {
            sompRaiseException(ev, SOMPERROR_FRAMEWORK_ERROR,
                                   SOMPERROR_ENC_DEC_NOT_FOUND);
         } /* endif */
      } /* endif */
   } /* endwhile */
   _sompFreeIterator(thisGroup, ev, hit);
   return(TRUE); /* return TRUE to indicate that we attempted to
                    write all objects in the group. */


#else
   /* Write only the given object. */
   /* Write the object only if it is stable and has been marked as
      "dirty" */
   if (_sompIsDirty(storeObj, ev)) {
      _sompPassivate(storeObj, ev); /* Tell object to prepare to be
                                       stored. */
      /* ...
         At this point, the IO Group Mgr must
         store anything about the object necessary in order to
         find it again in the group container
         ... */
```

```
      ed = _sompGetEncoderDecoder(storeObj, ev);
      if (ed) {
         /* Now, write the object data */
         _sompEDWrite(ed, ev, _sompGetMediaInterface(somSelf, ev),
                                                   storeObj);
         _sompFreeEncoderDecoder(storeObj, ev);
         _sompClearState(storeObj, ev, SOMP_STATE_DIRTY);
         /* ...
            any other unique processing
            ... */
      } else {
         sompRaiseException(ev, SOMPERROR_FRAMEWORK_ERROR,
                                SOMPERROR_ENC_DEC_NOT_FOUND);
      } /* endif */
   } /* endif */
   return(FALSE); /* return FALSE to indicate that we attempted to
                     write only the given object. */
#endif /* TMP_WRITE_ALL */
}

SOM_Scope SOMPPersistentObject  SOMLINK sompReadGroup(
                                  SOMPTemplate somSelf,
                                  Environment *ev,
                                  SOMPPersistentId objectID)
{
    SOMPIOGroup newIOGroup = NULL;
    SOMClass classObj;
    SOMObject thisPo = NULL;
    string objectClassName = "ttest"; /* template test class name */

    SOMPTemplateData *somThis = SOMPTemplateGetData(somSelf);
    SOMPTemplateMethodDebug("SOMPTemplate","sompReadGroup");

    /* ...
       Restore the objectClassName from the container the
       IO Group Mgr is managing and anything else about the object
       required for your IO Group Mgr. This will depend on what
       object metadata you store along with the object.

       You must provide some ability to map the given objectID
       into a SOM class name. Perhaps the class name is part of the
       persistent ID. For example, you might have persistent ID
       which includes not only the file name but the class name as
       well:
                MyIOGroupMgr:<filename>,<classname>:0

       objectClassName = ...;
       In this sample template, the class name is just set to
       "ttest" which will probably never be correct for your
       I/O Group Manager.

       ... */

   classObj = _somFindClass(SOMClassMgrObject,
                            SOM_IdFromString(objectClassName),
                            0, 0);
   if (classObj) { /* If SOM was able to find the class, instantiate
                      the object. */
      thisPo = _somNew(classObj);
      _sompInitGivenId(thisPo, ev, objectID);
```

```
      /* Indicate this object is unstable - i.e. not fully
         restored yet and that the object is not ”dirty” yet. */
      _sompClearState(thisPo, ev, SOMP_STATE_STABLE);
      _sompSetState(thisPo, ev, SOMP_STATE_UNSTABLE);
      _sompClearState(thisPo, ev, SOMP_STATE_DIRTY);
      newIOGroup = _sompGetIOGroup(thisPo, ev);
   } else {
      sompRaiseException(ev, SOMPERROR_FRAMEWORK_ERROR,
                             SOMPERROR_IOGROUP_NEWOBJ);
   } /* endif */

   return(thisPo);
}

SOM_Scope void  SOMLINK sompReadObjectData(SOMPTemplate somSelf,
                                        Environment *ev,
                                        SOMPPersistentObject thisPo)
{
    SOMPEncoderDecoderAbstract ed;
    SOMPPersistentId objectID;

    SOMPTemplateData *somThis = SOMPTemplateGetData(somSelf);
    SOMPTemplateMethodDebug(”SOMPTemplate”,”sompReadObjectData”);

    /* Locate the object in the group container via its ID and
       set up the media interface in preparation for invoking the
       sompEDRead method to read the object data.

       objectID = _sompGetPersistentId(thisPo, ev);

       ...

       */

    /* Note: the encoder/decoder class object must exist prior
        to a call to sompGetEncoderDecoder              */
    ed = _sompGetEncoderDecoder(thisPo, ev);
    if (ed) {
       _sompEDRead(ed, ev, _sompGetMediaInterface(somSelf, ev),
                   thisPo);
       _sompFreeEncoderDecoder(thisPo, ev);

       _sompClearState(thisPo, ev, SOMP_STATE_UNSTABLE);
       _sompSetState(thisPo, ev, SOMP_STATE_STABLE);
    } else {
       sompRaiseException(ev, SOMPERROR_FRAMEWORK_ERROR,
                              SOMPERROR_ENC_DEC_NOT_FOUND);
    } /* endif */
}

SOM_Scope void  SOMLINK sompDeleteObjectFromGroup(SOMPTemplate
somSelf, Environment *ev,
                SOMPPersistentId objectID)
{
    /* SOMPTemplateData *somThis = SOMPTemplateGetData(somSelf); */
    SOMPTemplateMethodDebug(”SOMPTemplate”,
                           ”sompDeleteObjectFromGroup”);

    /* This method has not been implemented in this template */


SOMPTemplate_parent_SOMPIOGroupMgrAbstract_sompDeleteObjectFromGroup
(somSelf,ev,objectID);
}
```

```
SOM_Scope boolean  SOMLINK sompGroupExists(SOMPTemplate somSelf,
Environment *ev,
                string IOInfo)
{
    SOMPTemplateData *somThis = SOMPTemplateGetData(somSelf);
    SOMPTemplateMethodDebug("SOMPTemplate","sompGroupExists");

    /* Determine whether the group given by IOInfo exists */
    /* This method has been implemented over optimistically
       in this template. */

    return (TRUE);
}

SOM_Scope boolean  SOMLINK sompObjectInGroup(SOMPTemplate somSelf,
Environment *ev,
                SOMPPersistentId objectID)
{
    SOMPTemplateData *somThis = SOMPTemplateGetData(somSelf);
    SOMPTemplateMethodDebug("SOMPTemplate","sompObjectInGroup");

    /* Determine whether the object with the given ID is exists in
       the group determined from the ID.
       This method has been implemented over optimistically in this
       template.
     */
    return (TRUE);
}

SOM_Scope boolean  SOMLINK sompMediaFormatOk(SOMPTemplate somSelf,
                                 Environment *ev,
                                 string mediaFormatName)
{
    SOMPTemplateData *somThis = SOMPTemplateGetData(somSelf);
    SOMPTemplateMethodDebug("SOMPTemplate","sompMediaFormatOk");

    /* Is the file being referred to by the caller one which we know
       how to read?
       This method has been implemented over optimistically in this
       template.
     */

    return (TRUE);
}

SOM_Scope void  SOMLINK somInit(SOMPTemplate somSelf)
{
    SOMPTemplateData *somThis = SOMPTemplateGetData(somSelf);
    SOMPTemplateMethodDebug("SOMPTemplate","somInit");

    /* Initialize instance variables */
    _mia = NULL; /* Initially, no media interface */
}

SOM_Scope void  SOMLINK somUninit(SOMPTemplate somSelf)
{
    Environment tev;
    SOMPTemplateData *somThis = SOMPTemplateGetData(somSelf);
    SOMPTemplateMethodDebug("SOMPTemplate","somUninit");

    /* Clean up resources */
    _sompFreeMediaInterface(somSelf, &tev);
}
```

An Example I/O Group Manager and Media Interface Implementation

The following final examples illustrate an implementation of a Media Interface subclass and an I/O Group Manager subclass that uses the new Media Interface.

In these examples, we show how to modify the SOM Persistence Framework so that you can store objects in an OS/2 INI file or “initialization file”. INI files are a convenient place to store information on the OS/2 system. The system typically uses these files to store configuration and startup information.

An OS/2 INI file consists of one or more named sections of data. Each section contains a set of key–value pairs. The section name and key are null terminated strings, while the value associated with the key is a variable length block of any data. When objects are stored using the example classes below, each class is stored in one section of the OS/2 INI file. Within the class section each key–value pair corresponds to the key part of an object’s persistent ID (retrieved via **sompGetGroupOffset**) and the value of an object’s instance data. Refer to the OS/2 Technical Library for more information on INI files and the OS/2 Profile Manager.

Note: AIX and Windows users may not be familiar with INI files, however the following implementation is still a useful illustration of how to build subclasses of the SOM Persistence Framework.

The Media Interface

Since we’ve decided to store our objects in a new type of file for which we don’t know the format, we must first provide the SOM Persistence Framework with an interface that does know the format of the file. To do that we must define a new Media Interface which includes calls to an API that can read/write INI files. We have a choice of subclassing from either **SOMPMediaInterfaceAbstract** or **SOMPFileMediaAbstract**. The supplied default Encoder/Decoder **SOMPAttrEncoderDecoder** makes use of the interface provided by the **SOMPFileMediaAbstract** class. To continue to take advantage of the supplied Encoder/Decoder we’ve chosen to implement a **SOMPFileMediaAbstract** interface to the INI files.

Ordinarily, you don’t stream data to an INI file. Each write for a key–value pair overwrites the previous write. The supplied Encoder/Decoder, however, expects to be able to stream data to its Media Interface. To maintain this ability, we introduce two new methods to our Media Interface. These are:

```
void sompBeginBlock(in string appName, in string key);

void sompEndBlock();
```

sompBeginBlock instructs the Media Interface to begin buffering all incoming write requests. sompEndBlock flushes all buffered data to the INI file. To support reading as well, sompBeginBlock first reads any data that matches the appName and key value. So, either reading or writing may follow sompBeginBlock.

Because all object data is buffered in memory, it is assumed that the objects stored are relatively small in size.

Here is the interface definition of the new Media Interface, called **SOMPIniMediaInterface**. Refer to the sample code in the SOMobjects Toolkit for a completed implementation of the class.

```
#ifndef prf_idl
#define prf_idl

#include <fmi.idl>
#include <somp.idl>
#include <somperrd.idl>

interface SOMPIniMediaInterface : SOMPAsciiMediaInterface

// This is the class definition for a media interface which
// reads/writes using the OS/2 profile API. (.INI files)
//
//
// An .INI file is not a stream oriented file. Data is passed to it
// in blocks, with each block labeled by an application name and key
// value.
//
// Since the SOM Persistence Framework makes many "read" or "write"
// calls to store one object, this class must collect all these
// requests into one final block request to the OS/2 profile API.
//
// It is assumed that this class is made use of in the following
// way:
//
//   /// IO Group Mgr ///
//
//   sompInitReadWrite(ini_file_name)
//   sompOpen
//   sompBeginBlock(app_name, key_string)
//
//   /// encoder/decoder ///
//
//       sompWrite...
//       sompWrite...
//       sompWrite...
//       ...
//
//   /// IO Group Mgr ///
//
//   sompEndBlock
//   sompClose
//
//
//   or:
//
//   /// IO Group Mgr ///
//
//   sompInitReadWrite(ini_file_name)
//   sompOpen
//   sompBeginBlock(app_name, key_string)
//
//   /// encoder/decoder ///
//
//       sompRead...
//       sompRead...
//       sompRead...
//       ...
//
```

```
//   /// IO Group Mgr ///
//
//   sompEndBlock
//   sompClose
//

{

   void sompBeginBlock(in string appName, in string key);
// Method to begin a block of data. Invoke after sompOpen.

   void sompEndBlock();
// Flushes buffered data to the profile.

   boolean sompBlockExists(in string appName, in string key);
// Check to see if the block named by the given appName and key
// exists. Returns TRUE if it exists. Invoke this after sompOpen
// but before sompBeginBlock.

#ifdef __SOMIDL__

implementation
{
   callstyle=idl;

   releaseorder: sompBeginBlock, sompEndBlock;
   dllname = "sompini.dll";

// Class Modifiers
   filestem = prf;

#ifdef __PRIVATE__
// Internal Instance variables
// Note: because the following OS/2 types are not defined as IDL
// types, warnings may be generated. This is ok, although you will
// not be able to store information about this data in the SOM
// Interface Repository.

#pragma notc 1

HAB      hab;         // Anchor-block handle
string   fileName;    // User-profile file name
HINI     hini;        // Handle to INI file
string   appName;     // App name passed to profile read/write
string   key;         // Key passed to profile read/write
string   block;       // Block of data to be read/written to the
profile
unsigned long readPtr;    // Position within the block to read
unsigned long writePtr;    // Position within the block to write
unsigned long blockSize; // Current size of the block
#endif
```

```
// Method Modifiers
    sompInitReadWrite      : override;
    sompInitReadOnly       : override;
    sompOpen               : override;
    sompClose              : override;
    sompSeekPosition       : override;
    sompSeekPositionRel    : override;
    sompGetOffset          : override;
    sompReadBytes          : override;
    sompWriteBytes         : override;
    sompWriteCharacter     : override;
    sompReadCharacter      : override;
    somInit                : override;
    somUninit              : override;
};
#endif /* __SOMIDL__ */
};
#endif  /* prf_idl */
```

The SOMPIniMediaInterface implementation

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>

#define PRFBLOCK_SIZE 1024

#ifdef __OS2__
#define INCL_WINWINDOWMGR    /* Or use INCL_WIN or INCL_PM. Also in
                                COMMON section */
#define INCL_WINSHELLDATA  /* Or use INCL_WIN or INCL_PM */
#include <os2.h>
#else
/* Stubs of OS/2 prf calls */
#include <prfstub.h>
#endif

#include <somp.h>
#include <string.h>
#include <memory.h>

#define SOMPIniMediaInterface_Class_Source
#include <prf.ih>

/* set the target string t with the given string g */
static string cpyString(string *t, string g)
{
    if (*t) SOMFree(*t);
    *t = (string)SOMMalloc(strlen(g) + 1);
    strcpy(*t, g);
    return(*t);
}

/*
 *  Method to begin a block of data
 */
```

```
SOM_Scope void  SOMLINK sompBeginBlock(
                SOMPIniMediaInterface somSelf,
                Environment *ev,
                string appName,
                string key)
{
    BOOL    fSuccess;   /* Success indicator */
    ULONG   dataLen = 0;
    SOMPIniMediaInterfaceData *somThis =
                        SOMPIniMediaInterfaceGetData(somSelf);
    SOMPIniMediaInterfaceMethodDebug("SOMPIniMediaInterface",
                                     "sompBeginBlock");

    if (_block) {
      SOMFree(_block);
    } /* endif */
    cpyString(&_appName, appName);
    cpyString(&_key, key);
    fSuccess = PrfQueryProfileSize(_hini, _appName, _key, &dataLen);
    if (dataLen == 0) {             /* No data by that name */
       _block = (string)SOMMalloc(PRFBLOCK_SIZE);
       _blockSize = PRFBLOCK_SIZE;
       memset((void*)_block, 0, (size_t)_blockSize);
       _readPtr = _blockSize;       /* nothing to read      */
       _writePtr = 0;
    } else {
       _block = (string)SOMMalloc(dataLen);
       fSuccess = PrfQueryProfileData(_hini, _appName, _key, _block,
                                      &dataLen);
       _blockSize = dataLen;
       _readPtr = 0;           /* can read from 0 to blockSize */
       _writePtr = 0;
    } /* endif */
}

/*
 *  Flushes buffered data to the profile
 */

SOM_Scope void  SOMLINK sompEndBlock(SOMPIniMediaInterface somSelf,
                                     Environment *ev)
{
    BOOL   fSuccess; /* Success indicator */
    SOMPIniMediaInterfaceData *somThis =
                       SOMPIniMediaInterfaceGetData(somSelf);
    SOMPIniMediaInterfaceMethodDebug("SOMPIniMediaInterface",
                                     "sompEndBlock");

    if (_writePtr) {
       fSuccess = PrfWriteProfileData(_hini, _appName, _key, _block,
                                      (ULONG)_writePtr);
    } /* endif */
}
```

```
SOM_Scope void  SOMLINK sompInitReadWrite(
                        SOMPIniMediaInterface somSelf,
                        Environment *ev,
                        string mediaInfo)
{
    SOMPIniMediaInterfaceData *somThis =
    SOMPIniMediaInterfaceGetData(somSelf);
    SOMPIniMediaInterfaceMethodDebug("SOMPIniMediaInterface",
                                     "sompInitReadWrite");

    cpyString(&_fileName, mediaInfo);
}

SOM_Scope void  SOMLINK sompInitReadOnly(
                        SOMPIniMediaInterface somSelf,
                        Environment *ev,
                        string mediaInfo)
{
    SOMPIniMediaInterfaceData *somThis =
                              SOMPIniMediaInterfaceGetData(somSelf);
    SOMPIniMediaInterfaceMethodDebug("SOMPIniMediaInterface",
                                     "sompInitReadOnly");
    _sompInitReadWrite(somSelf, ev, mediaInfo);
                                                  /* No readonly
                                                     support
                                                     for this version*/
}

static char userPrf[80]="os2.ini";
static char sysPrf[80]="os2sys.ini";
static PRFPROFILE p = {
   sizeof(userPrf), userPrf,
   sizeof(sysPrf), sysPrf };

/* Assume that if they give os2.ini then use USERPROFILE */
/* Assume that if they give os2sys.ini then use SYSTEMPROFILE */
SOM_Scope void  SOMLINK sompOpen(SOMPIniMediaInterface somSelf,
                                 Environment *ev)
{
    BOOL   fSuccess; /* Success indicator */
    SOMPIniMediaInterfaceData *somThis =
                             SOMPIniMediaInterfaceGetData(somSelf);
    SOMPIniMediaInterfaceMethodDebug("SOMPIniMediaInterface",
                                     "sompOpen");
```

```
    if (_fileName) {
       /* fSuccess = PrfQueryProfile(_hab, &p); */
#ifdef __OS2__
       if (stricmp(_fileName, userPrf)==0)
#else
       if (strcmp(_fileName, userPrf)==0)
#endif
       {
          _hini = HINI_USERPROFILE; /* No need to open */
       } else {
#ifdef __OS2__
          if (stricmp(_fileName, sysPrf)==0)
#else
          if (strcmp(_fileName, sysPrf)==0)
#endif
          {
             _hini = HINI_SYSTEMPROFILE;
          } else {
             _hini = PrfOpenProfile(_hab, _fileName);
          }
       }
    } /* endif */
}

SOM_Scope void  SOMLINK sompClose(SOMPIniMediaInterface somSelf,
                                  Environment *ev)
{
    BOOL  fSuccess;/* Success indicator */
    Environment tev;
    SOMPIniMediaInterfaceData *somThis =
                             SOMPIniMediaInterfaceGetData(somSelf);
    SOMPIniMediaInterfaceMethodDebug("SOMPIniMediaInterface",
                                     "sompClose");

    if (_hini) {
       fSuccess = PrfCloseProfile(_hini);
       _hini = NULL;
       _sompBeginBlock(somSelf, &tev, "$$SOMP$$", "key");
    } /* endif */
}

SOM_Scope void  SOMLINK sompSeekPosition(
                               SOMPIniMediaInterface somSelf,
                               Environment *ev,
                               long offset)
{
    SOMPIniMediaInterfaceData *somThis =
                              SOMPIniMediaInterfaceGetData(somSelf);
    SOMPIniMediaInterfaceMethodDebug("SOMPIniMediaInterface",
                                     "sompSeekPosition");

    _readPtr = offset;
}
```

```
SOM_Scope void  SOMLINK sompSeekPositionRel(
                        SOMPIniMediaInterface somSelf,
                        Environment *ev,
                        long offset)
{
    SOMPIniMediaInterfaceData *somThis =
                           SOMPIniMediaInterfaceGetData(somSelf);
    SOMPIniMediaInterfaceMethodDebug("SOMPIniMediaInterface",
                                  "sompSeekPositionRel");

    _readPtr += offset;
}

SOM_Scope long  SOMLINK sompGetOffset(SOMPIniMediaInterface somSelf,
                                      Environment *ev)
{
    SOMPIniMediaInterfaceData *somThis =
                           SOMPIniMediaInterfaceGetData(somSelf);
    SOMPIniMediaInterfaceMethodDebug("SOMPIniMediaInterface",
                                  "sompGetOffset");

    return((long)_readPtr);
}

SOM_Scope void  SOMLINK sompReadBytes(SOMPIniMediaInterface somSelf,
                                      Environment *ev,
                                      string byteStream,
                                      long length)
{
    long amountToRead = 0;
    SOMPIniMediaInterfaceData *somThis =
                           SOMPIniMediaInterfaceGetData(somSelf);
    SOMPIniMediaInterfaceMethodDebug("SOMPIniMediaInterface",
                                  "sompReadBytes");

    if (_blockSize) {
       amountToRead = ((_readPtr + length) <= _blockSize) ?
                      length :
                      _blockSize - _readPtr;

       if (amountToRead) {
          memcpy((void*)byteStream, (void*)(_block+_readPtr),
                                   (size_t)amountToRead);
          _readPtr += amountToRead;
       } else {
          /* Nothing to read */
       } /* endif */
    } else {
       /* Nothing to read */
    } /* endif */
}

SOM_Scope void  SOMLINK sompWriteBytes(
                     SOMPIniMediaInterface somSelf,
                     Environment *ev,
                     string byteStream,
                     long length)
{
    SOMPIniMediaInterfaceData *somThis =
                           SOMPIniMediaInterfaceGetData(somSelf);
    SOMPIniMediaInterfaceMethodDebug("SOMPIniMediaInterface",
                                  "sompWriteBytes");
```

```
    if (_blockSize) {
       if (_writePtr + length > _blockSize) {
          _blockSize += (PRFBLOCK_SIZE + length);
          _block = SOMRealloc(_block, _blockSize);
       } /* endif */
       memcpy((void*)(_block+_writePtr), (void*)byteStream,
                                         (size_t)length);
       _writePtr += length;
    } else {
       /* No place to write */
    } /* endif */

}

SOM_Scope void  SOMLINK sompWriteCharacter(
                        SOMPIniMediaInterface somSelf,
                        Environment *ev,
                        char c)
{
    SOMPIniMediaInterfaceData *somThis =
                              SOMPIniMediaInterfaceGetData(somSelf);
    SOMPIniMediaInterfaceMethodDebug("SOMPIniMediaInterface",
                                     "sompWriteCharacter");

   _sompWriteBytes(somSelf, ev,  &c, 1);
}

SOM_Scope void  SOMLINK sompReadCharacter(
                        SOMPIniMediaInterface somSelf,
                        Environment *ev,
                        string c)
{
    SOMPIniMediaInterfaceData *somThis =
                              SOMPIniMediaInterfaceGetData(somSelf);
    SOMPIniMediaInterfaceMethodDebug("SOMPIniMediaInterface",
                                     "sompReadCharacter");

   _sompReadBytes(somSelf, ev, c, 1);
}

SOM_Scope void  SOMLINK somInit(SOMPIniMediaInterface somSelf)
{
    static int firsttime = 1;
    Environment *ev;
    SOMPIniMediaInterfaceData *somThis =
                              SOMPIniMediaInterfaceGetData(somSelf);

SOMPIniMediaInterfaceMethodDebug("SOMPIniMediaInterface","somInit");

    if (firsttime) {
       _hab = WinInitialize(0L);
       firsttime = 0;
    } /* endif */

    _fileName = NULL;
    _hini = NULL;
    ev = SOM_CreateLocalEnvironment();
    _sompBeginBlock(somSelf, ev, "$$SOMP$$", "key");
    SOM_DestroyLocalEnvironment(ev);
}
```

```
SOM_Scope void  SOMLINK somUninit(SOMPIniMediaInterface somSelf)
{
    SOMPIniMediaInterfaceData *somThis =
                                SOMPIniMediaInterfaceGetData(somSelf);
    SOMPIniMediaInterfaceMethodDebug(”SOMPIniMediaInterface”,
                                       ”somUninit”);

    if (_block) {
       SOMFree(_block);
    } /* endif */
}

/*
 *  Method to check for the existence of a block of data.
 */
SOM_Scope boolean  SOMLINK sompBlockExists(
                SOMPIniMediaInterface somSelf,
                Environment *ev,
                string appName,
                string key)
{
    BOOL    fSuccess;   /* Success indicator */
    ULONG   dataLen = 0;
    SOMPIniMediaInterfaceData *somThis =
                                SOMPIniMediaInterfaceGetData(somSelf);
    SOMPIniMediaInterfaceMethodDebug(”SOMPIniMediaInterface”,
                                       ”sompBeginBlock”);

    fSuccess = PrfQueryProfileSize(_hini, appName, key, &dataLen);
    if (!fSuccess || dataLen == 0) { /* No data by that name */
       return(FALSE);
    } else {
       return(TRUE);
    } /* endif */
}
```

The I/O Group Manager

Whenever you build a new Media Interface, you need to build a new I/O Group Manager. To build our new I/O Group Manager, SOMPIni, we started with the template described in “Template for an I/O Group Manager” earlier in this appendix. The persistent object ID used when an object is initialized determines which I/O Group Manager is instantiated when an object is stored/restored. The name part of the ID (set via **sompSetIOGroupName** or indirectly via somutSetIdString) is meaningful only to the I/O Group Manager class. In our implementation, the name is used to specify the INI file in which to store our objects. An example of an object ID for our I/O Group Manager where the objects are stored in the os2.ini file would be:

```
SOMPIni:os2.ini:0
```

In order for this I/O Group Manager to store objects via the Media Interface we created above, we’ll need to provide the Media Interface with an appName and key for the sompBeginBlock method. In our implementation, we make the appName equal to the object’s class name and the key value equal to the offset part of the object ID. This part of the ID can be returned via the **sompGetGroupOffset** method on the object ID.

When an object is to be restored, the I/O Group Manager is only presented with an object ID. Since our ID does not indicate what class of object is to be instantiated during the restore, we must provide a mapping between ID and class name. We’ve chosen to implement this by reserving a special block of INI data for ID and class name information. It would have also been possible to implement this by deciding that all IDs for our I/O Group Manager contain the class name of the object. For example:

```
SOMPIni:os2.ini,PersistentStudent:0
```

In this example, we've chosen the earlier alternative.

Writing an object comes down to:

- getting the object's Encoder/Decoder object.
- doing sompBeginBlock() to start the ID–to–class–name mapping block
- storing the class name and ID
- ending the mapping block with sompEndBlock()
- calling the **sompEDWrite**() method of the Encoder/Decoder
- freeing the Encoder/Decoder
- ending the class data block with sompEndBlock()

Here is the interface definition of the new I/O Group Manager, SOMPIni. Refer to the sample code in the installed SOMobjects Developer Toolkit for the complete implementation of the class.

```
#ifndef prfgm_idl
#define prfgm_idl

#include <iogma.idl>
#include <somp.idl>
#include <somperrd.idl>
#include <sompstad.idl>

interface SOMPIni : SOMPIOGroupMgrAbstract

// This is an IO Group Manager class for OS/2 .INI files. It uses
// the SOMPIniMediaInterface to store objects in OS/2 .INI files.
//
//
// This class expects object ID's of the form:
//
//    SOMPIni:<INI_file_name>:<offset_key>
//
// where:
//
//    <INI_file_name> is the name of the INI file to read/write to.
//    <INI_file_name> == os2.ini means use the USER PROFILE.
//    <INI_file_name> == os2sys.ini means use the SYSTEM PROFILE.
//

{

#ifdef __SOMIDL__

implementation
{
   callstyle=idl;

   dllname="sompini.dll";

// Internal Instance Variables
   SOMPMediaInterfaceAbstract mia;
```

```
// Method Modifiers
   sompNewMediaInterface: override;
   sompGetMediaInterface: override;
   sompFreeMediaInterface: override;
   sompInstantiateMediaInterface: override;
   sompWriteGroup: override;
   sompReadGroup: override;
   sompReadObjectData: override;
   sompDeleteObjectFromGroup: override;
   sompGroupExists: override;
   sompObjectInGroup: override;
   sompMediaFormatOk: override;
   somInit: override;
   somUninit: override;

};
#endif /* __SOMIDL__ */

};

#endif  /* prfgm_idl */
```

The SOMPIni Group Manager implementation

The differences between this I/O Group Manager and the template one it was started with are highlighted.

```
#include <somp.h>
#include <stdlib.h>
#include <stdio.h>
#include <iogrp.h>
#include <prf.h>

#define SOMPIni_Class_Source
#include <prfgm.ih>

#define MAPPING_APPNAME "$SOMP_KEY2CLASS$"

SOM_Scope void  SOMLINK sompNewMediaInterface(SOMPIni somSelf,
                                    Environment *ev,
                                    string IOInfo)
{
    SOMPIniData *somThis = SOMPIniGetData(somSelf);
    SOMPIniMethodDebug("SOMPIni","sompNewMediaInterface");

    if (_mia == NULL) {
       _mia = _sompInstantiateMediaInterface(somSelf, ev);
       _sompInitReadWrite(_mia, ev, IOInfo);
       if (ev->_major == NO_EXCEPTION) {
          _sompOpen(_mia, ev);
       } /* endif */
    } /* endif */
}

SOM_Scope SOMPMediaInterfaceAbstract  SOMLINK
            sompGetMediaInterface(SOMPIni somSelf, Environment *ev)
{
    SOMPIniData *somThis = SOMPIniGetData(somSelf);
    SOMPIniMethodDebug("SOMPIni","sompGetMediaInterface");

    return (_mia);
}
```

```
SOM_Scope void  SOMLINK sompFreeMediaInterface(SOMPIni somSelf,
                                               Environment *ev)
{
    SOMPIniData *somThis = SOMPIniGetData(somSelf);
    SOMPIniMethodDebug("SOMPIni","sompFreeMediaInterface");

    if (_mia) {
       _sompClose(_mia, ev);
       _somFree(_mia);
       _mia = NULL;
    } /* endif */
}

SOM_Scope SOMPMediaInterfaceAbstract  SOMLINK
                      sompInstantiateMediaInterface(SOMPIni somSelf,
                                                    Environment *ev)
{
    SOMPMediaInterfaceAbstract mia = NULL;
    SOMPIniData *somThis = SOMPIniGetData(somSelf);
    SOMPIniMethodDebug("SOMPIni","sompInstantiateMediaInterface");

    /* mia = <mediaInterfaceClassName>New(); */
    mia = SOMPIniMediaInterfaceNew();
    return(mia);
}

SOM_Scope boolean SOMLINK sompWriteGroup(SOMPIni somSelf,
Environment *ev,
                SOMPPersistentObject storeObj)
{
    SOMPIteratorHandle hit;
    SOMObject thisPo;
    SOMPIOGroup thisGroup;
    SOMPEncoderDecoderAbstract ed;
    long key;
    char keyString[SOMPMAXIDSIZE];

    SOMPIniData *somThis = SOMPIniGetData(somSelf);
    SOMPIniMethodDebug("SOMPIni","sompWriteGroup");

/* Write all objects grouped with the given object */
#define TMP_WRITE_ALL 1
#ifdef TMP_WRITE_ALL
    thisGroup = _sompGetIOGroup(storeObj, ev);

    hit = _sompNewIterator(thisGroup, ev);

    /* Get each of the objects from the group.
       If it should be stored, then instantiate an Encoder/Decoder
       object and invoke its sompEDWrite method. */
    while ( ((thisPo = _sompNextObjectInGroup(thisGroup, ev,
             hit))!=NULL) &&
            ev->_major == NO_EXCEPTION) {
```

```
      if (_sompIsDirty(thisPo, ev)) {
         _sompPassivate(thisPo, ev); /* Tell object to prepare to be
                                        stored. */
         ed = _sompGetEncoderDecoder(thisPo, ev);
         /* Now, write the object data */
         if (ed) {
            key = _sompGetGroupOffset(_sompGetPersistentId(thisPo,
                                       ev), ev);
            sprintf(keyString, "%d", key);
            /* First, store the class name associated with the */
            /* object ID so that upon object restoration, the  */
            /* name of the class associated with this ID can be*/
            /* determined.                                     */
            _sompBeginBlock(_mia, ev, MAPPING_APPNAME, keyString);
            _sompWriteString(_mia, ev, _somGetClassName(thisPo));
            _sompWriteString(_mia, ev,
                             _sompGetEncoderDecoderName(thisPo, ev));
            _sompEndBlock(_mia, ev);

            /* Now, store the object data...               */
            _sompBeginBlock(_mia, ev, _somGetClassName(thisPo),
                            keyString);
            _sompEDWrite(ed, ev, _sompGetMediaInterface(somSelf,
                                 ev), thisPo);
            _sompFreeEncoderDecoder(thisPo, ev);
            _sompClearState(thisPo, ev, SOMP_STATE_DIRTY);
            _sompEndBlock(_mia, ev);
         } else {
            /* No encoder/decoder object could be instantiated, */
            /* perhaps the <encoder/decoderClassName>NewClass   */
            /* has not be executed.                             */
            sompRaiseException(ev, SOMPERROR_FRAMEWORK_ERROR,
                                   SOMPERROR_ENC_DEC_NOT_FOUND);
         } /* endif */
         /* ...
            any other unique processing
            ... */
      } /* endif */
   } /* endwhile */
   _sompFreeIterator(thisGroup, ev, hit);
   return(TRUE);
#endif /* TMP_WRITE_ALL */
}

SOM_Scope SOMPPersistentObject  SOMLINK sompReadGroup(
                        SOMPIni somSelf,
                        Environment *ev,
                        SOMPPersistentId objectID)
{
    SOMPIOGroup newIOGroup = NULL;
    SOMClass classObj;
    SOMObject thisPo = NULL;
    long key;
    char keyString[SOMPMAXIDSIZE];
    char objectClassName[SOMPMAXIDSIZE];
    char encoderDecoderClassName[SOMPMAXIDSIZE];

    SOMPIniData *somThis = SOMPIniGetData(somSelf);
    SOMPIniMethodDebug("SOMPIni","sompReadGroup");
```

```
   /* Restore class names                                          */
   /* Note: you could use encoderDecoderClassName data here to  */
   /*       set the object's encoderDecoder class name. We are  */
   /*       using the default, so are ignoring this information.*/
   key = _sompGetGroupOffset(objectID, ev);
   sprintf(keyString, "%d", key);
   _sompBeginBlock(_mia, ev, MAPPING_APPNAME, keyString);
   _sompReadStringToBuffer(_mia, ev, objectClassName,
                           SOMPMAXIDSIZE);
   _sompReadStringToBuffer(_mia, ev, encoderDecoderClassName,
                           SOMPMAXIDSIZE);
   _sompEndBlock(_mia, ev);

   classObj = _somFindClass(SOMClassMgrObject,
                            SOM_IdFromString(objectClassName),
                            0, 0);
   if (classObj) { /* If SOM was able to find the class, instantiate
                      the object. */
      thisPo = _somNew(classObj);
      _sompInitGivenId(thisPo, ev, objectID);
      /*
         ...       */
      /* Indicate this object is unstable - i.e. not fully restored
         yet and that the object is not "dirty" yet.            */
      _sompClearState(thisPo, ev, SOMP_STATE_STABLE);
      _sompSetState(thisPo, ev, SOMP_STATE_UNSTABLE);
      _sompClearState(thisPo, ev, SOMP_STATE_DIRTY);
      newIOGroup = _sompGetIOGroup(thisPo, ev);
   } else {
      sompRaiseException(ev, SOMPERROR_FRAMEWORK_ERROR,
                             SOMPERROR_IOGROUP_NEWOBJ);
   } /* endif */

   return(thisPo);
}

SOM_Scope void  SOMLINK sompReadObjectData(SOMPIni somSelf,
                Environment *ev,
                SOMPPersistentObject thisPo)
{
    SOMPEncoderDecoderAbstract ed;
    SOMPPersistentId objectID;
    long key;
    char keyString[SOMPMAXIDSIZE];

    SOMPIniData *somThis = SOMPIniGetData(somSelf);
    SOMPIniMethodDebug("SOMPIni","sompReadObjectData");

    /* Locate the object in the group container via its ID and
       set up the media interface in preparation for invoking the
       sompEDRead method to read the object data.
       objectID = _sompGetPersistentId(thisPo, ev);
```

```
        ...        */
     ed = _sompGetEncoderDecoder(thisPo, ev);
     if (ed) {
        key = _sompGetGroupOffset(_sompGetPersistentId(thisPo, ev),
                                   ev);
        sprintf(keyString, "%d", key);
        _sompBeginBlock(_mia, ev, _somGetClassName(thisPo),
                        keyString);
        /* Restore class names                                    */
        /* Note: you could use encoderDecoderClassName data here to*/
        /*       set the object's encoderDecoder class name. We are*/
        /*       using the default, so are ignoring this          */
        /*       information.                                     */
        _sompEDRead(ed, ev, _sompGetMediaInterface(somSelf, ev),
                    thisPo);
        _sompFreeEncoderDecoder(thisPo, ev);
        _sompEndBlock(_mia, ev);
        _sompClearState(thisPo, ev, SOMP_STATE_UNSTABLE);
        _sompSetState(thisPo, ev, SOMP_STATE_STABLE);
     } else {
        /* No encoder/decoder object could be instantiated.       */
        sompRaiseException(ev, SOMPERROR_FRAMEWORK_ERROR,
                               SOMPERROR_ENC_DEC_NOT_FOUND);
     } /* endif */

}

SOM_Scope void  SOMLINK sompDeleteObjectFromGroup(SOMPIni somSelf,
Environment *ev,
                SOMPPersistentId objectID)
{
    /* SOMPIniData *somThis = SOMPIniGetData(somSelf); */
    SOMPIniMethodDebug("SOMPIni","sompDeleteObjectFromGroup");

    /* Not implemented                                       */

SOMPIni_parent_SOMPIOGroupMgrAbstract_sompDeleteObjectFromGroup(somS
elf,ev,objectID);
}

SOM_Scope boolean  SOMLINK sompGroupExists(SOMPIni somSelf,
Environment *ev,
                string IOInfo)
{
    SOMPIniData *somThis = SOMPIniGetData(somSelf);
    SOMPIniMethodDebug("SOMPIni","sompGroupExists");

    /* Very optimistic implementation                      */
    return (TRUE);
}

SOM_Scope boolean  SOMLINK sompObjectInGroup(SOMPIni somSelf,
Environment *ev,
                SOMPPersistentId objectID)
{
    long key;
    char keyString[SOMPMAXIDSIZE];
    SOMPIniData *somThis = SOMPIniGetData(somSelf);
    SOMPIniMethodDebug("SOMPIni","sompObjectInGroup");
```

```
    key = _sompGetGroupOffset(objectID, ev);
    sprintf(keyString, "%d", key);
    return(_sompBlockExists(_mia, ev, MAPPING_APPNAME, keyString));
}

SOM_Scope boolean  SOMLINK sompMediaFormatOk(SOMPIni somSelf,
Environment *ev,
                string mediaFormatName)
{
    SOMPIniData *somThis = SOMPIniGetData(somSelf);
    SOMPIniMethodDebug("SOMPIni","sompMediaFormatOk");

    /* Very optimistic implementation                              */
    return (TRUE);
}

SOM_Scope void  SOMLINK somInit(SOMPIni somSelf)
{
    SOMPIniData *somThis = SOMPIniGetData(somSelf);
    SOMPIniMethodDebug("SOMPIni","somInit");

    _mia = NULL; /* Initially, no media interface */
}

SOM_Scope void  SOMLINK somUninit(SOMPIni somSelf)
{
    Environment tev;
    SOMPIniData *somThis = SOMPIniGetData(somSelf);
    SOMPIniMethodDebug("SOMPIni","somUninit");

    _sompFreeMediaInterface(somSelf, &tev);
}
```

Appendix E. Implementing Sockets Subclasses

Contents

Appendix E. Implementing Sockets Subclasses

Distributed SOM (DSOM) and the Replication Framework require basic message services for inter-process communications. The Event Management Framework must be integrated with the same communication services in order to handle communications events.

To maximize their portability to a wide variety of local area network transport protocols, the DSOM, Replication, and Event Management Frameworks have been written to use a *common communications interface*, which is implemented by one or more SOM class libraries using available local protocols.

The common communications interface is based on the “sockets” interface used with TCP/IP, since its interface and semantics are fairly widespread and well understood. The IDL interface is named **Sockets**. There is no implementation associated with the **Sockets** interface by default; specific protocol implementations are supplied by subclass implementations.

Note: The **Sockets** classes supplied with the SOMobjects Developer Toolkit and Enabler packages are *only* intended to support the DSOM, Replication, and Event Management Frameworks. These class implementations are *not* intended for general application usage.

Available **Sockets** subclasses for SOMobjects products are as follows:

- For AIX:
  TCPIPSockets class for TCP/IP,
  IPXSockets class for NetWare IPX/SPX, and
  NBSockets class for NetBIOS.
- For OS/2 and Windows:
  TCPIPSockets class (a) for TCP/IP for Windows or (b) for TCP/IP 1.2.1 on OS/2,
  TCPIPSockets32 class for TCP/IP 2.0 on OS/2 only (see Note below),
  IPXSockets class for NetWare IPX/SPX, and
  NBSockets class for NetBIOS.

Note: The **TCPIPSockets32** class gives greater performance over the **TCPIPSockets** class on OS/2, but requires the 32–bit version of TCP/IP (version 2.0) rather than the 16–bit version of TCP/IP (version 1.2.1).

Application developers may need to develop their own **Sockets** subclass if the desired transport protocol or product version is not one of those supported by the SOMobjects run-time packages. This appendix explains how to approach the implementation of a **Sockets** subclass, if necessary. Warning: this may be a non-trivial exercise!

Sockets IDL interface

The base **Sockets** interface is expressed in IDL in the file **somssock.idl**, listed below. There is a one-to-one mapping between TCP/IP socket APIs and the methods defined in the **Sockets** interface.

Please note the following:

- The semantics of each **Sockets** method must be that of the corresponding TCP/IP call. Currently, only Internet address family (AF_INET) addresses are used by the frameworks.

  (The TCP/IP sockets API is not documented as part of the SOMobjects Developer Toolkit. The implementor is referred to the programming references for IBM TCP/IP for AIX or OS/2, or to similar references that describe the sockets interface for TCP/IP.)
- Data types, constants, and macros which are part of the **Sockets** interface are defined in a C include file, **soms.h**. This file is supplied with the SOMobjects Toolkit, and is not shown in this manual.

- The **Sockets** interface is expressed in terms of a 32-bit implementation.
- Some of the method parameters and return values are expressed using pointer types, for example:

  ```
  hostent *somsGethostent ();
  ```

  This has been done to map TCP/IP socket interfaces as directly as possible to their IDL equivalent. (Use of strict CORBA IDL was not a primary goal for the **Sockets** interface, since it is only used internally by the frameworks.)
- The **Sockets** class and its subclasses are *single instance* classes.

Following is a listing of the file **somssock.idl**. Each socket call is briefly described with a comment.

```
// 96F8647, 96F8648 (C) Copyright IBM Corp. 1992, 1993
// All Rights Reserved
// Licensed Materials - Property of IBM

#ifndef somssock_idl
#define somssock_idl

#include <somobj.idl>
#include <snglicls.idl>

interface Sockets : SOMObject
{
    //# The following typedefs are fully defined in <soms.h>.
    typedef SOMFOREIGN sockaddr;
    #pragma modifier sockaddr : impctx="C", struct;
    typedef SOMFOREIGN iovec;
    #pragma modifier iovec : impctx="C", struct;
    typedef SOMFOREIGN msghdr;
    #pragma modifier msghdr : impctx="C", struct;
    typedef SOMFOREIGN fd_set;
    #pragma modifier fd_set : impctx="C", struct;
    typedef SOMFOREIGN timeval;
    #pragma modifier timeval : impctx="C", struct;
    typedef SOMFOREIGN hostent;
    #pragma modifier hostent : impctx="C", struct;
    typedef SOMFOREIGN servent;
    #pragma modifier servent : impctx="C", struct;
    typedef SOMFOREIGN in_addr;
    #pragma modifier in_addr : impctx="C", struct;

    long somsAccept (in long s, out sockaddr name, out long namelen);
    // Accept a connection request from a client.

    long somsBind (in long s, inout sockaddr name, in long namelen);
    // Binds a unique local name to the socket with descriptor s.

    long somsConnect (in long s, inout sockaddr name,
                      in long namelen);
    // For streams sockets, attempts to establish a connection
    // between two sockets.  For datagram sockets, specifies the
    // socket's peer.

    hostent *somsGethostbyaddr (in char *addr, in long addrlen,
                                in long domain);
    // Returns a hostent structure for the host address specified on
    // the call.
```

```
hostent *somsGethostbyname (in string name);
// Returns a hostent structure for the host name specified on
// the call.

hostent *somsGethostent ();
// Returns a pointer to the next entry in the hosts file.

unsigned long somsGethostid ();
// Returns the unique identifier for the current host.

long somsGethostname (in string name, in long namelength);
// Retrieves the standard host name of the local host.

long somsGetpeername (in long s, out sockaddr name,
                      out long namelen);
// Gets the name of the peer connected to socket s.

servent *somsGetservbyname (in string name, in string protocol);
// Retrieves an entry from the /etc/services file using the
// service name as a search key.

long somsGetsockname (in long s, out sockaddr name,
                      out long namelen);
// Stores the current name for the socket specified by the s
// parameter into the structure pointed to by the name
// parameter.

long somsGetsockopt (in long s, in long level, in long optname,
                     in char *optval, out long option);
// Returns the values of socket options at various protocol
// levels.

unsigned long somsHtonl (in unsigned long a);
// Translates an unsigned long integer from host-byte order to
// network-byte order.

unsigned short somsHtons (in unsigned short a);
// Translates an unsigned short integer from host-byte order to
// network-byte order.

long somsIoctl (in long s, in long cmd, in char *data,
                in long length);
// Controls the operating characteristics of sockets.

unsigned long somsInet_addr (in string cp);
// Interprets character strings representing numbers expressed
// in standard '.' notation and returns numbers suitable for use
// as internet addresses.

unsigned long somsInet_lnaof (in in_addr addr);
// Breaks apart the internet address and returns the local
// network address portion.

in_addr somsInet_makeaddr (in unsigned long net,
                           in unsigned long lna);
// Takes a network number and a local network address and
// constructs an internet address.

unsigned long somsInet_netof (in in_addr addr);
// Returns the network number portion of the given internet
// address.
```

```
unsigned long somsInet_network (in string cp);
// Interprets character strings representing numbers expressed
// in standard '.' notation and returns numbers suitable for use
// as network numbers.

string somsInet_ntoa (in in_addr addr);
// Returns a pointer to a string expressed in the dotted-decimal
// notation.

long somsListen (in long s, in long backlog);
// Creates a connection request queue of length backlog to queue
// incoming connection requests, and then waits for incoming
// connection requests.

unsigned long somsNtohl (in unsigned long a);
// Translates an unsigned long integer from network-byte order
// to host-byte order.

unsigned short somsNtohs (in unsigned short a);
// Translates an unsigned short integer from network-byte order
// to host-byte order.

long somsReadv (in long s, inout iovec iov, in long iovcnt);
// Reads data on socket s and stores it in a set of buffers
// described by iov.

long somsRecv (in long s, in char *buf, in long len,
               in long flags);
// Receives data on streams socket s and stores it in buf.

long somsRecvfrom (in long s, in char *buf, in long len,
    in long flags, out sockaddr name, out long namelen);
// Receives data on datagram socket s and stores it in buf.

long somsRecvmsg (in long s, inout msghdr msg, in long flags);
// Receives messages on a socket with descriptor s and stores
// them in an array of message headers.

long somsSelect (in long nfds, inout fd_set readfds,
                 inout fd_set writefds, inout fd_set exceptfds,
                 inout timeval timeout);
// Monitors activity on a set of different sockets until a
// timeout expires, to see if any sockets are ready for reading
// or writing, or if an exceptional condition is pending.

long somsSend (in long s, in char *msg, in long len,
               in long flags);
// Sends msg on streams socket s.

long somsSendmsg (in long s, inout msghdr msg, in long flags);
// Sends messages passed in an array of message headers on a
// socket with descriptor s.

long somsSendto (in long s, inout char msg, in long len,
               in long flags, inout sockaddr to, in long tolen);
// Sends msg on datagram socket s.

long somsSetsockopt (in long s, in long level, in long optname,
                     in char *optval, in long optlen);
// Sets options associated with a socket.
```

```
    long somsShutdown (in long s, in long how);
    // Shuts down all or part of a full-duplex connection.

    long somsSocket (in long domain, in long type,
                     in long protocol);
    // Creates an endpoint for communication and returns a socket
    // descriptor representing the endpoint.

    long somsSoclose (in long s);
    // Shuts down socket s and frees resources allocated to the
    // socket.

    long somsWritev (in long s, inout iovec iov, in long iovcnt);
    // Writes data on socket s.  The data is gathered from the
    // buffers described by iov.

    attribute long serrno;
    // Used to pass error numbers.

#ifdef __SOMIDL__
    implementation
    {
    releaseorder:
        somsAccept, somsBind, somsConnect, somsGethostbyaddr,
        somsGethostbyname, somsGethostent, somsGethostid,
        somsGethostname, somsGetpeername, somsGetsockname,
        somsGetsockopt, somsHtonl, somsHtons, somsIoctl,
        somsInet_addr, somsInet_lnaof, somsInet_makeaddr,
        somsInet_netof, somsInet_network, somsInet_ntoa,
        somsListen, somsNtohl, somsNtohs, somsReadv,
        somsRecv, somsRecvfrom, somsRecvmsg, somsSelect,
        somsSend, somsSendmsg, somsSendto, somsSetsockopt,
        somsShutdown, somsSocket, somsSoclose, somsWritev,
        _set_serrno, _get_serrno, somsGetservbyname;

    //# Class modifiers
    callstyle=idl;
    metaclass = SOMMSingleInstance;
    majorversion=1; minorversion=1;
    dll="soms.dll";
    };
#endif /* __SOMIDL__ */
};
#endif  /* somssock_idl */
```

IDL for a Sockets subclass

Sockets subclasses inherit their entire interface from **Sockets.** All methods are overridden.

For example, here is a listing of the **TCPIPSockets** IDL description.

```
// 96F8647, 96F8648 (C) Copyright IBM Corp. 1992, 1993
// All Rights Reserved
// Licensed Materials - Property of IBM

#ifndef tcpsock_idl
#define tcpsock_idl

#include <somssock.idl>
#include <snglicls.idl>
```

```
interface TCPIPSockets : Sockets
{
#ifdef __SOMIDL__
    implementation
    {
    //# Class modifiers
    callstyle=idl;
    majorversion=1; minorversion=1;
    dllname="somst.dll";
    metaclass=SOMMSingleInstance;

    //# Method modifiers
    somsAccept: override;
    somsBind: override;
    somsConnect: override;
    somsGethostbyaddr: override;
    somsGethostbyname: override;
    somsGethostent: override;
    somsGethostid: override;
    somsGethostname: override;
    somsGetpeername: override;
    somsGetservbyname: override;
    somsGetsockname: override;
    somsGetsockopt: override;
    somsHtonl: override;
    somsHtons: override;
    somsIoctl: override;
    somsInet_addr: override;
    somsInet_lnaof: override;
    somsInet_makeaddr: override;
    somsInet_netof: override;
    somsInet_network: override;
    somsInet_ntoa: override;
    somsListen: override;
    somsNtohl: override;
    somsNtohs: override;
    somsReadv: override;
    somsRecv: override;
    somsRecvfrom: override;
    somsRecvmsg: override;
    somsSelect: override;
    somsSend: override;
    somsSendmsg: override;
    somsSendto: override;
    somsSetsockopt: override;
    somsShutdown: override;
    somsSocket: override;
    somsSoclose: override;
    somsWritev: override;
    _set_serrno: override;
    _get_serrno: override;
    };
#endif /* __SOMIDL__ */
};

#endif  /* tcpsock_idl */
```

Implementation considerations

- Only the AF_INET address family must be supported. That is, the DSOM, Replication, and Event Manager frameworks all use Internet addresses and port numbers to refer to specific sockets.
- On OS/2, the SOMobjects run-time libraries were built using the C Set/2 32-bit compiler. If the underlying subclass implementation uses a 16-bit subroutine library, conversion of the method call arguments may be required. (This mapping of arguments is often referred to as "thunking.")
- **Sockets** subclasses to be used in multi-threaded environments should be made thread-safe. That is, it is possible that concurrent threads may make calls on the (single) **Sockets** object, so data structures must be protected within critical regions, as appropriate.
- Valid values for the **serrno** attribute are defined in the file **soms.h**. The subclass implementation should map local error numbers into the appropriate corresponding **Sockets** error numbers.

Example code

The following code fragment shows an example of the implementation of the **somsBind** method of the **TCPIPSockets** subclass, for both AIX and OS/2. The sample illustrates that, for TCP/IP, the implementation is basically a one-to-one mapping of **Sockets** methods onto TCP/IP calls. For other transport protocols, the mapping from the socket abstraction to the protocol's API may be more difficult.

For AIX, the mapping from **Sockets** method to TCP/IP call is trivial.

```
SOM_Scope long  SOMLINK somsBind(TCPIPSockets somSelf,
                                 Environment *ev,
                                 long s, Sockets_sockaddr* name,
                                 long namelen)
{
    long rc;

    TCPIPSocketsMethodDebug("TCPIPSockets","somsBind");

    rc = (long) bind((int)s, name, (int)namelen);

    if (rc == -1)
        __set_serrno(somSelf, ev, errno);

    return rc;
}
```

On OS/2, however, the TCP/IP Release 1.2.1 library is a 16-bit library. Consequently, many of the method calls require conversion ("thunking") of 32-bit parameters into 16-bit parameters, before the actual TCP/IP calls can be invoked. For example, the function prototype for the **somsBind** method is defined as:

```
SOM_Scope long  SOMLINK somsBind(TCPIPSockets somSelf,
                                 Environment *ev,
                                 long s, Sockets_sockaddr* name,
                                 long namelen);
```

whereas the file **socket.h** on OS/2 declares the **bind** function with the following prototype:

```
short _Far16 _Cdecl bind(short /*s*/, void * _Seg16 /*name*/,
                         short /*len*/);
```

In this case, the pointer to the “name” structure, passed as a 32-bit address, cannot be used directly in the **bind** call: a 16-bit address must be passed instead. This can be accomplished by dereferencing the 32-bit pointer provided by the “name” parameter in the **somsBind** call, copying the caller’s Sockets_sockaddr structure into a local structure (“name16”), and then passing the address of the local structure (“&name16”) as a 16-bit address in the **bind** call.

```
SOM_Scope long  SOMLINK somsBind(TCPIPSockets somSelf,
                                 Environment *ev,
                                 long s, Sockets_sockaddr* name,
                                 long namelen)
{
    long rc;
    Sockets_sockaddr name16;

    TCPIPSocketsMethodDebug(”TCPIPSockets”,”somsBind”);

    /* copy user’s parameter into a local structure */
    memcpy ((char *)&name16, (char *)((sockaddr32 *)name), namelen);
    rc = (long) bind((short)s, (void *)&name16, (short)namelen);

    if (rc == –1)
        __set_serrno(somSelf, ev, tcperrno());

    return rc;
}
```

For Windows, a developer would follow the OS/2 example for implementing the **bind** function with 16–bit addresses (but using the **IPXSockets** class for NetWare IPX/SPX or the **NBSockets** class for NetBIOS, rather than the **TCPIPSockets** class).

Appendix F. emitcom: An Emitter of COM Interfaces

Contents

Appendix F. emitcom: An Emitter of COM Interfaces

The **emitcom** emitter is a program that creates a binding for a SOM class so that the class can be used in the context of COM, Microsoft's component interface model. That is, the binding exports COM-style interfaces so that a SOM class can be used from OLE 2.0 programs. The generated COM interface is aggregatable. The **emitcom** emitter generates all the files necessary to build a DLL for the binding. In addition, **emitcom** can generate COM bindings for ancestor classes.

'emitcom' Syntax

The **emitcom** command is issued as follows:

emitcom *<filestem>* *<comstem>*

where: *<filestem>* is the prefix name of a SOM IDL file (*<filestem>***.idl**), and
<comstem> is the prefix for the corresponding COM binding files.

Execution of 'emitcom'

For the IDL file *<filestem>***.idl**, **emitcom** creates a set of files that compose an interface (or usage binding) that gives an OLE 2.0 program access to the SOM class described in *<filestem>***.idl**. The following files are created: *<comstem>***.mak**, *<comstem>***.xh**, *<comstem>***.cpp**, *<comstem>***.def**, and *<comstem>***.reg**. Once **emitcom** has run, issue the commands:

nmake –f *<comstem>***.mak**	to create a DLL and LIB; and
regedit /s *<comstem>***.reg**	to register the DLL with the REG.DAT database.

The COM interface generated by **emitcom** is the SOM class's interface. That is, the interface contains the union of the methods of the SOM class and all of its ancestors.

The COM interface is generated in C++; *<comstem>***.cpp** is the implementation file and *<comstem>***.xh** is a header file for users of the interface. Because SOM is language neutral, it does not matter what language is used to implement the SOM class.

The *<comstem>***.def** file is used by the linker to make the DLL. In generating a makefile (*<comstem>***.mak**), **emitcom** makes the following decision:

- If the *<filestem>***.idl** file contains a **dllname** modifier, the associated *<dllname>***.lib** file is used in the link statement. If there is no **dllname** modifier in the IDL file, then the link statement is generated with *<filestem>***.obj**. (See also "Modifier statements" in Chapter 4, "Implementing Classes in SOM," of the *SOMobjects Developer Toolkit Users Guide*.)
- If *<filestem>***.obj** is not desired, one can always edit the *<comstem>***.mak**.

The *<comstem>***.reg** file contains the information for registering the COM interface to the SOM class in the registration database. The DLL file name that is used in *<comstem>***.reg** is *<comstem>***.dll**; if this DLL is to be named otherwise, you must edit the *<comstem>***.reg** file.

The COM interface for the SOM class named *<className>* is defined in *<comstem>***.xh**. The interface is implemented as a C++ class named *<className>***COMIntf**. To use the SOM class in a program, one must include the header *<comstem>***.xh** and create instances of the C++ class *<className>***COMIntf**, which creates instances of the SOM class.

Interface Identifiers

The *<filestem>***.idl** file must give the class identifier and the interface identifier needed for registration. This is done with two new modifiers:

CLSID_*<className>* and
IID_*<className>*

where *<className>* is the name of the SOM class for which a COM interface is being generated. The various forms of the modifiers are as follows:

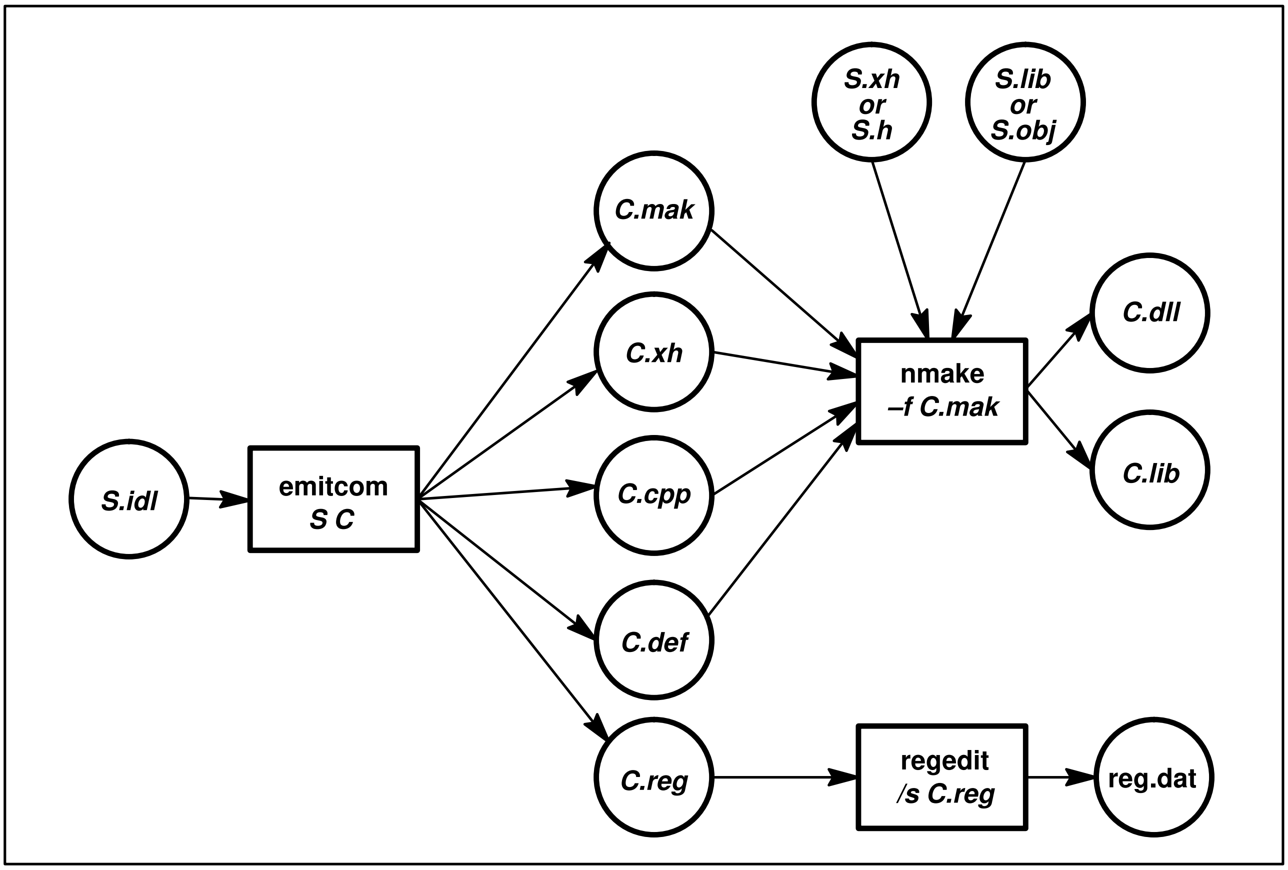
CLSID_*<className>* = *<guid1>*;
IID_*<className> = <guid2>;*
IID_*<parentClassName1> = <guid3>;*
IID_*<parentClassName2>;*
IID_*<SOMObjectsToolkitClassName>;*

The first two forms are mandatory, because the class and interface identifiers for the SOM class must be specified. The third form is used to specify an interface identifier for a parent class. However, if an interface identifier is specified in the IDL of the parent, the fourth form should be used. The fifth form is used for parent classes that are part of the SOMObjects Toolkit. Note that the third, fourth, and fifth forms are used only when an interface to the parent is to be aggregated into the COM binding.

Each ancestor of a SOM class provides an interface to instances of that class. Therefore, each of these may be aggregated into the COM binding. This is indispensable in the case where the instance is to be passed to code that was created for the ancestor interface (that is, code that uses the COM binding generated from the ancestor's IDL). In such cases, the caller must coerce the instance interface by calling QueryInterface before passing the instance into the code created for the ancestor.

User Procedure

The following diagram depicts the total process, where *<filestem>* is ***S*** and *<comstem>* is ***C***.



In summary, you will perform the following steps:

1. Add CLSID and IID to *<filestem>***.idl** with the modifers:
   CLSID_*<className>*
   IID_*<className>*

   There is an example below.

2. Run the **"emitcom** *<filestem> <comstem>***"** command to produce the files:
   *<comstem>***.mak** *<comstem>***.xh** *<comstem>***.cpp**
   *<comstem>***.def** *<comstem>***.reg**

   Note: *<comstem>***.mak** is generated with the value of the **dllname** SOM IDL modifier or *<filestem>***.obj** in the LINK command.

3. Run **"nmake –f** *<comstem>***.mak"** to produce the files:
   *<comstem>***.lib** *<comstem>***.dll**

4. Run **"regedit /s** *<comstem>***.reg"** to update reg.dat (\windows\reg.dat). Remember to update the DLL location in *<comstem>***.reg** if necessary.

5. Install the header (*<comstem>***.xh**) and library (*<comstem>***.dll** and *<comstem>***.lib**) in the required directory.

The Generated Interface

Suppose the SOM class in *<filestem>***.idl** is named *X*. The COM interface generated by **emitcom** in the *<comstem>***.xh** file then appears as follows.

```
#include "<filestem>.xh"

DEFINE_GUID (CLSID_X,  <class identifier>);
DEFINE_GUID (IID_X, <interface identifier>);

class XCOMIntf : IUnknown
{
 public:
    XCOMIntf( LPUNKNOWN );          // constructor

    STDMETHOD(QueryInterface)(REFIID riid, void FAR* FAR* ppv);
    STDMETHOD_(ULONG, AddRef)(void);
    STDMETHOD_(ULONG, Release)(void);

    // SOM methods
    < all methods supported by X >

};

class XCOMFactory : public IClassFactory
{
  public:
    XCOMFactory();

    STDMETHOD(QueryInterface)(REFIID riid, void FAR* FAR* ppv);
    STDMETHOD_(ULONG, AddRef)(void);
    STDMETHOD_(ULONG, Release)(void);

    STDMETHOD(CreateInstance)(IUnknown FAR* punkOuter,
                              REFIID riid,
                              void FAR* FAR* ppv);
    STDMETHOD(LockServer)(BOOL fLock);
};
```

There is a C++ class named *XCOMIntf* that contains the three *IUnknown* methods and all of the methods that the SOM class *X* supports (that is, any method defined in the *X* SOM class or any of its ancestor classes).

There is one constructor for *XCOMIntf* which takes an LPUNKNOWN parameter that is the *pUnkOuter* of the controlling interface in the case that *XCOMIntf* is part of an aggregate. If the interface is not part of an aggregate, the constructor should be called with a NULL value.

Customizing the *<comstem>*.mak

The *<comstem>***.mak** file is used to create a DLL that implements COM interface. The file is designed to be invoked from a makefile. There are two macro parameters in *<comstem>***.mak** that can be set: OBJS and LIBS. The first is used to indicate any other object files that are to be linked into the DLL. The second is used to specify any other libraries on which the DLL depends.

In addition, when the environment variable COMDEBUG is set to 1, the *<comstem>***.dll** is compiled with the debugger options.

Example

As an example, the standard SOM "Hello" sample has been modified to generate a COM binding for the "Hello" class. The full text of this modified example is also among the SOM samples. The following is a modified IDL file for the "Hello" SOM sample program that can be used to generate a COM interface. Note that the "Hello" sample SOM class is implemented in C (not C++), yet the COM binding is implemented in C++.

```
#include <somobj.idl>
interface Hello : SOMObject
/* this is a simple class that demonstrates how to define
 * the interface to a new class of objects in SOM IDL.
 */
{
        string sayHello();
        // This method returns the string "Hello, World!".
#ifdef __SOMIDL__
implementation
 {
 releaseorder: sayHello;
 CLSID_Hello = "12345678-abcd-1234-1234-123456789012";
 IID_Hello =   "01234567-0123-cdef-0123-012345678901";
 };
#endif
};
```

Next is a fragment of a main program that uses the COM interface generated by **emitcom**. Note that, although this looks like using a SOM class with the C++ bindings, it actually is an example of using a COM interface. That is, *HelloCOMClass* is an implementation of a COM interface that supports both the *IUnknown* methods and all the methods of the "Hello" SOM class.

```
HelloCOMIntf *pintf;
HRESULT hr;
LPCLASSFACTORY  pHelloFactory;

switch (message){
    case WM_CREATE:
        hr = CoGetClassObject(CLSID_Hello,
                              CLSCTX_INPROC_SERVER,
                              NULL,
                              IID_IClassFactory,
                              (void FAR* FAR*)& pHelloFactory);
        if ( SUCCEEDED(hr) ) {
                pHelloFactory->CreateInstance(NULL,
                                       IID_Hello,
                                         (void FAR* FAR*)&pintf );
                pHelloFactory->Release();
                }
        else {
                PostQuitMessage(2);
                }
        return 0;
    case WM_PAINT:
        hdc = BeginPaint (hwnd, &ps) ;
        GetClientRect (hwnd, &rect) ;
        strcpy(sBuf, pintf->sayHello(somGetGlobalEnvironment()));
        DrawText (hdc, sBuf, -1, &rect,
                       DT_SINGLELINE | DT_CENTER | DT_VCENTER);
        EndPaint (hwnd, &ps) ;
        return 0 ;
    case WM_DESTROY:
        PostQuitMessage (0) ;
        return 0 ;
    }
```

Following is an example of the main procedure for the preceding message loop.

```
#include <comhello.xh>

long FAR PASCAL _export WndProc (HWND, UINT, UINT, LONG) ;

int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
                      LPSTR lpszCmdParam, int nCmdShow)
{
    static char szAppName[] = ”Hello” ;
    HWND        hwnd ;
    MSG         msg ;
    WNDCLASS    wndclass ;

    HRESULT hr;
    hr = CoInitialize( NULL );
    if ( !SUCCEEDED(hr) ) {
        exit(1) }

    if (!hPrevInstance){
        wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
        wndclass.lpfnWndProc   = WndProc ;
        wndclass.cbClsExtra    = 0 ;
        wndclass.cbWndExtra    = 0 ;
        wndclass.hInstance     = hInstance ;
        wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
        wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
        wndclass.hbrBackground = GetStockObject (LTGRAY_BRUSH) ;
        wndclass.lpszMenuName  = NULL ;
        wndclass.lpszClassName = szAppName ;

        RegisterClass (&wndclass) ; }
    hwnd = CreateWindow (szAppName, // window class name
            ”Hello Program”,         // window caption
            WS_OVERLAPPEDWINDOW,     // window style
            CW_USEDEFAULT,           // initial x position
            CW_USEDEFAULT,           // initial y position
            CW_USEDEFAULT,           // initial x size
            CW_USEDEFAULT,           // initial y size
            NULL,                    // parent window handle
            NULL,                    // window menu handle
            hInstance,               // program instance handle
            NULL) ;                  // creation parameters

    ShowWindow (hwnd, nCmdShow) ;
    UpdateWindow (hwnd) ;

    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;}

    CoUninitialize( );

    return msg.wParam ;
}
```

Limitations

The following are known limitations at the current time:

- *<filestem>***.idl** cannot contain more than one interface nor can it contain IDL modules.
- **emitcom** creates the following temporary files: *<filestem>***.cmm**, *<filestem>***.cmh**, *<filestem>***.cmc**, *<filestem>***.cmd**, and *<filestem>***.reg**. The **emitcom** emitter should not be run in a directory where you have files with these names (when **emitcom** runs, it overwrites these files). Note that *<filestem>* is the first parameter to **emitcom**.
- *<comstem>***.mak** is for Microsoft's **nmake**; *<comstem>***.mak** expects the C++ compiler to be named **cl**. This makefile uses the temporary file *<comstem>***.lrf**.

Glossary

Note: In the following definitions, words shown in *italics* are terms for which separate glossary entries are also defined.

abstract class
A *class* that is not designed to be instantiated, but serves as a *base class* for the definition of subclasses. Regardless of whether an abstract class inherits *instance data* and *methods* from *parent classes*, it will always introduce methods that must be *overridden* in a *subclass,* in order to produce a class whose objects are semantically valid.

affinity group An array of *class objects* that were all registered with the *SOMClassMgr* object during the dynamic loading of a *class*. Any class is a member of at most one affinity group.

ancestor class
A *class* from which another class inherits *instance methods, attributes,* and *instance variables*, either directly or indirectly. A direct descendant of an ancestor class is called a *child class, derived class,* or *subclass.* A direct ancestor of a class is called a *parent class, base class*, or *superclass*.

aggregate type
A user-defined data type that combines basic types (such as, char, short, float, and so on) into a more complex type (such as structs, arrays, strings, sequences, unions, or enums).

apply stub A *procedure* corresponding to a particular *method* that accepts as arguments: the *object* on which the method is to be invoked, a pointer to a location in memory where the method's result should be stored, a pointer to the method's procedure, and the method's arguments in the form of a va_list. The apply stub extracts the arguments from the va_list, invokes the method with its arguments, and stores its result in the specified location. Apply stubs are registered with class objects when instance methods are defined, and are invoked using the somApply function. Typically, *implementations* that *override* somDispatch call somApply to invoke a method on a va_list of arguments.

attribute A specialized syntax for declaring "set" and "get" methods. Method names corresponding to attributes always begin with "_set_" or "_get_". An attribute name is declared in the body of the *interface statement* for a class. Method procedures for get/set methods are automatically defined by the *SOM Compiler* unless an attribute is declared as "noget/noset". Likewise, a corresponding *instance variable* is automatically defined unless an attribute is declared as "nodata". IDL also supports "readonly" attributes, which specify only a "get" method. (Contrast an attribute with an *instance variable.*)

auxiliary class data structure
A structure provided by the SOM API to support efficient static access to *class*-specific information used in dealing with SOM *objects*. The structure's name is <*className*>CClassData. Its first component (parentMtab) is a list of *parent-class method tables* (used to support efficient parent method calls). Its second component (instanceDataToken) is the *instance token* for the class (generally used to locate the *instance data* introduced by *method procedures* that implement *methods* defined by the class).

base class See *parent class.*

behavior (of an object)
The *methods* that an *object* responds to. These methods are those either introduced or inherited by the *class* of the object. See also *state.*

bindings Language-specific macros and procedures that make implementing and using SOM classes more convenient. These bindings offer a convenient interface to SOM that is tailored to a particular programming language. The *SOM Compiler* generates binding files for C and C++. These binding files include an *implementation template* for the class and two header files, one to be included in the class's implementation file and the other in client programs.

BOA (basic object adapter) class
A CORBA *interface* (represented as an *abstract class* in DSOM), which defines generic *object-adapter (OA)* methods that a *server* can use to register itself and its *objects* with an *ORB (object request broker)*. See also *SOMOA (SOM object adapter) class.*

callback A user-provided procedure or method to the Event Management Framework that gets invoked when a registered event occurs. (See also *event*).

casted dispatching
A form of method dispatching that uses *casted method resolution*; that is, it uses a designated ancestor class of the actual *target object's* class to determine what procedure to call to execute a specified method.

casted method resolution
A *method resolution* technique that uses a *method procedure* from the *method table* of an ancestor of the *class* of an *object* (rather than using a procedure from the method table of the object's own class).

child class A class that inherits *instance methods, attributes,* and *instance variables* directly from another class, called the *parent class, base class*, or *superclass,* or indirectly from an *ancestor class.* A child class may also be called a *derived class* or *subclass.*

class A way of categorizing *objects* based on their behavior (the *methods* they support) and shape (memory layout). A class is a definition of a generic object. In SOM, a class is also a special kind of object that can manufacture other objects that all have a common shape and exhibit similar behavior. The specification of what comprises the shape and behavior of a set of objects is referred to as the "definition" of a class. New classes are defined in terms of existing classes through a technique known as *inheritance*. See also *class object.*

class variable *Instance data* of a *class object*. All instance data of an *object* is defined (through either introduction or *inheritance*) by the object's class. Thus, class variables are defined by *metaclasses*.

class data structure
A structure provided by the SOM API to support efficient static access to *class*-specific information used in dealing with SOM *objects*. The structure's name is <*className*>ClassData. Its first component (classObject) is a pointer to the corresponding *class object*. The remaining components (named after the *instance methods* and *instance variables*) are *method tokens* or *data tokens*, in order as specified by the class's implementation. Data tokens are only used to support data (public and private) introduced by classes declared using *OIDL*; IDL *attributes* are supported with method tokens.

class manager
An *object* that acts as a run-time registry for all SOM *class objects* that exist within the current process and which assists in the dynamic loading and unloading of class libraries. A class implementor can define a customized class manager by subclassing *SOMClassMgr* class to replace the SOM-supplied SOMClassMgrObject. This is done to augment the functionality of the default class-management registry (for example, to coordinate the automatic quiescing and unloading of classes).

class method (Also known as *factory method* or *constructor*.) A class method is a *method* that a *class object* responds to (as opposed to an *instance method*). A class method that class <X> responds to is provided by the *metaclass* of class <X>. Class methods are executed without requiring any *instances* of class <X> to exist, and are frequently used to create instances of the class.

class object The run-time *object* representing a SOM *class.* In SOM, a class object can perform the same behavior common to all *objects*, inherited from *SOMObject*.

client code (Or *client program* or *client*.) An application program, written in the programmer's preferred language, which invokes *methods* on *objects* that are *instances* of SOM *classes*. In DSOM, this could be a program that invokes a method on a remote object.

constructor See *class method*.

context expression
An optional expression in a method's IDL declaration, specifying identifiers whose value (if any) can be used during SOM's *method resolution* process and/or by the *target object* as it executes the *method procedure*. If a context expression is specified, then a related Context parameter is required when the method is invoked. (This Context parameter is an *implicit parameter* in the IDL specification of the method, but it is an explicit parameter of the method's procedure.) No SOM-supplied methods require context parameters.

CORBA The Common Object Request Broker Architecture established by the Object Management Group. IBM's *Interface Definition Language* used to describe the *interface* for SOM classes is fully compliant with CORBA standards.

data token A value that identifies a specific *instance variable* within an *object* whose *class inherits* the instance variable (as a result of being derived, directly or indirectly, from the class that introduces the instance variable). An object and a data token are passed to the SOM run-time procedure, somDataResolve, which returns is a pointer to the specific instance variable corresponding to the data token. (See also *instance token.*)

derived class See *subclass* and *subclassing.*

derived metaclass
(Or *SOM-derived metaclass*.) A *metaclass* that SOM creates automatically (often even when the *class* implementor specifies an explicit metaclass) as needed to ensure that, for any code that executes without *method-resolution* error on an *instance* of a given class, the code will similarly execute without method-resolution error on instances of any *subclass* of the given class. SOM's ability to derive such metaclasses is a fundamental necessity in order to ensure binary compatibility for client programs despite any subsequent changes in class *implementations*.

descriptor (Or *method descriptor*.) An ID representing the identifier of a *method* definition or an *attribute* definition in the Interface Repository. The IR definition contains information about the method's return type and the type of its arguments.

directive A message (a pre-defined character constant) received by a *replica* from the Replication Framework. Indicates a potential failure situation.

dirty object A persistent *object* that has been modified since it was last written to persistent storage.

dispatch-function resolution
Dispatch-function resolution is the slowest, but most flexible, of the three *method-resolution* techniques SOM offers. Dispatch functions permit method resolution to be based on arbitrary rules associated with an *object*'s *class*. Thus, a class implementor has complete freedom in determining how methods invoked on its *instances* are resolved. See also *dispatch method* and *dynamic dispatching.*

dispatch method
A *method* (such as somDispatch or somClassDispatch) that is invoked (and passed an argument list and the ID of another method) in order to determine the appropriate *method procedure* to execute. The use of dispatch methods facilitates *dispatch-function resolution* in SOM applications and enables method invocation on remote objects in DSOM applications. See also *dynamic dispatching*.

dynamic dispatching
Method dispatching using *dispatch-function resolution*; the use of dynamic *method resolution* at run time. See also *dispatch-function resolution* and *dynamic method.*

Dynamic Invocation Interface (DII)
The *CORBA*-specified *interface*, implemented in DSOM, that is used to dynamically build requests on remote *objects*. Note that DSOM applications can also use the somDispatch method for *dynamic method* calls when the object is remote. See also *dispatch method.*

dynamic method
A method that is not declared in the IDL *interface statement* for a *class* of *objects*, but is added to the *interface* at run time, after which *instances* of the class (or of its *subclasses*) will respond to the registered dynamic method. Because dynamic methods are not declared, *usage bindings* for SOM classes cannot support their use; thus, *offset method resolution* is not available. Instead, *name-lookup* or *dispatch-function method resolution* must be used to invoke dynamic methods. (There are currently no known uses of dynamic methods by any SOM applications.) See also *method* and *static method.*

encapsulation
An object-oriented programming feature whereby the implementation details of a class are hidden from client programs, which are only required to know the *interface* of a *class* (the signatures of its *methods* and the names of its *attributes*) in order to use the class's methods and attributes.

encoder/decoder
In the Persistence Framework, a *class* that knows how to read/write the persistent object format of a *persistent object*. Every persistent object is associated with an Encoder/Decoder, and an encoder/decoder object is created for each *attribute* and *instance variable*. An Encoder/Decoder is supplied by the Persistence Framework by default, or an application can define its own.

entry class In the Emitter Framework, a *class* that represents some syntactic unit of an *interface* definition in the *IDL source file.*

Environment parameter
A *CORBA*-required parameter in all *method procedures*, it represents a memory location where exception information can be returned by the *object* of a method invocation. [Certain methods are exempt (when the class contains a modifier of callstyle=oidl), to maintain upward compatibility for client programs written using an earlier release.]

emitter Generically, a program that takes the output from one system and converts the information into a different form. Using the Emitter Framework, selected output from the *SOM Compiler* (describing each syntactic unit in an *IDL source file*) is transformed and formatted according to a user-defined template. Example emitter output, besides the implementation template and language bindings, might include reference documentation, class browser descriptions, or "pretty" printouts.

event The occurrence of a condition, or the beginning or ending of an activity that is of interest to an application. Examples are elapse of a time interval, sending or receiving of a message, and opening or closing a file. (See also *event manager* and *callback*.)

event manager (EMan)
The chief component of the Event Management Framework that registers interest in various *events* from calling modules and informs them through *callbacks* when those events occur.

factory method See *class method*.

ID See *somId.*

IDL source file
A user-written .idl file, expressed using the syntax of the *Interface Definition Language* (IDL), which describes the *interface* for a particular *class* (or classes, for a *module*). The IDL source file is processed by the *SOM Compiler* to generate the *binding files* specific to the programming languages of the class implementor and the client application. (This file may also be called the "IDL file," the "source file," or the "interface definition file.")

implementation
(Or *object implementation*.) The specification of what *instance variables* implement an *object*'s *state* and what *procedures* implement its *methods* (or *behaviors*). In DSOM, a remote object's implementation is also characterized by its *server implementation* (a program).

Implementation Repository
A database used by DSOM to store the implementation definitions of DSOM *servers*.

implementation statement
An optional declaration within the body of the *interface* definition of a *class* in a SOM *IDL source file,* specifying information about how the class will be implemented (such as, version numbers for the class, overriding of inherited methods, or type of method resolution to be supported by particular methods). This statement is a SOM-unique statement; thus, it must be preceded by the term "#ifdef __SOMIDL__" and followed by "#endif". See also *interface declaration.*

implementation template
A template file containing *stub procedures* for *methods* that a *class* introduces or *overrides*. The implementation template is one of the *binding files* generated by the *SOM Compiler* when it processes the *IDL source file* containing class *interface declarations.* The class implementor then customizes the *implementation,* by adding language-specific code to the *method procedures*.

implicit method parameter

A *method* parameter that is not included in the IDL *interface* specification of a method, but which is a parameter of the *method's* *procedure* and which is required when the method is invoked from a *client program*. Implicit parameters include the required *Environment* parameter indicating where exception information can be returned, as well as a *Context* parameter, if needed.

incremental update

A revision to an *implementation template* file that results from reprocessing of the *IDL source file* by the *SOM Compiler*. The updated implementation file will contain new *stub procedures*, added comments, and revised *method prototypes* reflecting changes made to the *method* definitions in the IDL specification. Importantly, these updates do not disturb existing code that the class implementor has defined for the prior method procedures.

inheritance The technique of defining one *class* (called a *subclass, derived class,* or *child class*) as incremental differences from another class (called the *parent class, base class, superclass,* or *ancestor class*). From its parents, the subclass inherits variables and *methods* for its *instances*. The subclass can also provide additional *instance variables* and methods. Furthermore, the subclass can provide new procedures for implementing inherited methods. The subclass is then said to *override* the parent class's methods. An overriding method procedure can elect to call the parent class's *method procedure*. (Such a call is known as a *parent method call*.)

inheritance hierarchy

The sequential relationship from a root class to a subclass, through which the subclass inherits *instance methods, attributes,* and *instance variables* from all of its ancestors, either directly or indirectly. The root class of all SOM classes is SOMObject.

instance (Or *object instance* or just *object*.) A specific object, as distinguished from a *class* of objects. See also *object.*

instance method

A method valid for an object *instance* (as opposed to a *class method*, which is valid for a *class object*). An instance method that an object responds to is defined by its class or inherited from an ancestor class.

instance token

A *data token* that identifies the first *instance variable* among those introduced by a given *class*. The somGetInstanceToken *method* invoked on a *class object* returns that class's instance token.

instance variables

(Or, *instance data*.) Variables declared for use within the *method procedures* of a *class*. An instance variable is declared within the body of the *implementation statement* in a SOM *IDL source file*. An instance variable is "private" to the class and should not be accessed by a client program. (Contrast an instance variable with an *attribute.*)

interface The information that a *client* must know to use a *class* — namely, the names of its *attributes* and the signatures of its *methods*. The interface is described in a formal language (the *Interface Definition Language*, IDL) that is independent of the programming language used to implement the class's methods.

interface declaration
(Or *interface statement.*) The statement in the *IDL source file* that specifies the name of a new class and the names of its *parent class*(es). The "body" of the interface declaration defines the *signature* of each new *method* and any *attribute*(s) associated with the class. In SOM IDL, the body may also include an *implementation statement* (where *instance variables* are declared or a *modifier* is specified, for example to *override* a method).

Interface Definition Language (IDL)
The formal language (independent of any programming language) by which the *interface* for a *class* of *objects* is defined in a .idl file, which the *SOM Compiler* then interprets to create an *implementation template* file and *binding* files. SOM's Interface Definition Language is fully compliant with standards established by the Object Management Group's Common Object Request Broker Architecture (*CORBA*).

Interface Repository (IR)
The database that SOM optionally creates to provide persistent storage of objects representing the major elements of *interface* definitions. Creation and maintenance of the IR is based on information supplied in the *IDL source file.* The SOM IR Framework supports all interfaces described in the *CORBA* standard.

Interface Repository Framework
A set of *classes* that provide *methods* whereby executing programs can access the persistent objects of the *Interface Repository* to discover everything known about the programming *interfaces* of SOM classes.

macro An alias for executing a sequence of hidden instructions; in SOM, typically the means of executing a command known within a *binding file* created by the *SOM Compiler*.

metaclass A *class* whose *instances* are classes. In SOM, any class descended from *SOMClass* is a metaclass. The *methods* a class inherits from its metaclass are sometimes called *class methods* (in Smalltalk) or *factory methods* (in Objective-C) or *constructors*. See also *class method.*

metaclass incompatibility
A situation where a *subclass* does not include all of the *class variables* or respond to all of the *class methods* of its *ancestor classes*. This situation can easily arise in *OOP* systems that allow programmers to explicitly specify *metaclasses*, but is not allowed to occur in SOM. Instead, SOM automatically prevents this by creating and using *derived metaclasses* whenever necessary.

method A combination of a *procedure* and a name, such that many different procedures can be associated with the same name. In object-oriented programming, invoking a method on an *object* causes the object to execute a specific *method procedure*. The process of determining which method procedure to execute when a method is invoked on an object is called *method resolution*. (The *CORBA* standard uses the term "operation" for method invocation). SOM supports two different kinds of methods: static methods and dynamic methods. See also *static method* and *dynamic method.*

method descriptor See *descriptor*.

method ID A number representing a zero-terminated string by which SOM uniquely represents a *method* name. See also *somId.*

method procedure

A function or procedure, written in an arbitrary programming language, that implements a *method* of a *class*. A method procedure is defined by the class implementor within the *implementation template* file generated by the *SOM Compiler.*

method prototype

A *method* declaration that includes the types of the arguments. Based on method definitions in an *IDL source file*, the *SOM Compiler* generates method prototypes in the *implementation template*. A class implementor uses the method prototype as a basis for writing the corresponding *method procedure* code. The method prototype also shows all arguments and their types that are required to invoke the method from a *client program*.

method resolution

The process of selecting a particular *method procedure*, given a *method* name and an object *instance*. The process results in selecting the particular function/procedure that implements the abstract method in a way appropriate for the designated object. SOM supports a variety of method-resolution mechanisms, including *offset method resolution, name-lookup resolution,* and *dispatch-function resolution*.

method table A table of pointers to the *method procedures* that implement the *methods* that an *object* supports. See also *method token.*

method token A value that identifies a specific *method* introduced by a *class*. A method token is used during *method resolution* to locate the *method procedure* that implements the identified method. The two basic method-resolution procedures are somResolve (which takes as arguments an *object* and a method token, and returns a pointer to a procedure that implements the identified method on the given object) and somClassResolve (which takes as arguments a *class* and a method token, and returns a pointer to a procedure that implements the identified method on an instance of the given class).

modifier Any of a set of statements that control how a *class*, an *attribute*, or a *method* will be implemented. Modifiers can be defined in the *implementation statement* of a SOM *IDL source file*. The implementation statement is a SOM-unique extension of the *CORBA* specification. [User-defined modifiers can also be specified for use by user-written emitters or to store information in the *Interface Repository*, which can then be accessed via methods provided by the *Interface Repository Framework*.]

module The organizational structure required within an *IDL source file* that contains *interface declarations* for two (or more) classes that are not a class–metaclass pair. Such *interfaces* must be grouped within a module declaration.

multiple inheritance

The situation in which a *class* is derived from (and inherits *interface* and *implementation* from) multiple parent classes.

name-lookup method resolution

Similar to the *method resolution* techniques employed by Objective-C and Smalltalk. It is significantly slower than *offset resolution*. Name-lookup resolution, unlike offset resolution, can be used when the name of the method to be invoked is not known until run time, or the method is added to the class interface at run time, or the name of the class introducing the method is not known until run time.

naming scope See *scope*.

object (Or *object instance* or just *instance*.) An entity that has *state* (its data values) and *behavior* (its *methods*). An object is one of the elements of data and function that programs create, manipulate, pass as arguments, and so forth. An object is a way to encapsulate state and behavior. *Encapsulation* permits many aspects of the *implementation* of an object to change without affecting client programs that depend on the object's behavior. In SOM, objects are created by other objects called *classes*.

object adapter (OA)
A *CORBA* term denoting the primary interface a *server implementation* uses to access *ORB* functions; in particular, it defines the mechanisms that a server uses to interact with DSOM, and vice versa. This includes server activation/ deactivation, dispatching of *methods*, and authentication of the *principal* making a call. The basic object adapter described by CORBA is defined by the *BOA (basic object adapter) abstract class*; DSOM's primary object adapter implementation is provided by the *SOMOA* (*SOM Object Adapter*) *class*.

object definition See *class*.

object implementation See *implementation.*

object instance See *instance* and *object*.

object reference
A *CORBA* term denoting the information needed to reliably identify a particular *object*. This concept is implemented in DSOM with a *proxy object* in a *client* process, or a *SOMDObject* in a *server* process. See also *proxy object* and *SOMDObject.*

object request broker (ORB) See *ORB*.

offset method resolution
The default mechanism for performing *method resolution* in SOM, because it is the fastest (nearly as fast as an ordinary procedure call). It is roughly equivalent to the C++ "virtual function" concept. Using offset method resolution requires that the name of the *method* to be invoked must be known at compile time, the name of the *class* that introduces the method must be known at compile time (although not necessarily by the programmer), and the method to be invoked must be a *static method*.

OIDL The original language used for declaring SOM *classes*. The acronym stands for Object Interface Definition Language. OIDL is still supported by SOM release 2, but it does not include the ability to specify *multiple inheritance* classes.

one-copy serializable
The consistency property of the Replication Framework which states that the concurrent execution of *methods* on a *replicated object* is equivalent to the serial execution of those same methods on a nonreplicated object.

OOP An acronym for "object-oriented programming."

operation See *method.*

operation logging
In the Replication Framework, a technique for maintaining consistency among *replicas* of a replicated object, whereby the execution of a *method* that updates the *object* is repeated at the site of each replica.

ORB (object request broker)
A *CORBA* term designating the means by which *objects* transparently make requests (that is, invoke *methods*) and receive responses from objects, whether they are local or remote. With SOMobjects Developer Toolkit and Runtimes, this functionality is implemented in the DSOM Framework. Thus, the DSOM (Distributed SOM) system is an ORB. See also *BOA (basic object adapter) class* and *SOMOA (SOM object adapter) class.*

override (Or *overriding method.*) The technique by which a *class* replaces (redefines) the *implementation* of a *method* that it inherits from one of its *parent classes*. An overriding method can elect to call the parent class's *method procedure* as part of its own implementation. (Such a call is known as a *parent method call.*)

parent class A *class* from which another class inherits *instance methods*, *attributes*, and *instance variables*. A parent class is sometimes called a *base class* or *superclass*.

parent method call
A technique where an *overriding method* calls the *method procedure* of its *parent class* as part of its own *implementation*.

persistent object
An *object* whose *state* can be preserved beyond the termination of the *process* that created it. Typically, such objects are stored in files.

polymorphism
An object-oriented programming feature that may take on different meanings in different systems. Under various definitions of polymorphism, (a) a *method* or *procedure* call can be executed using arguments of a variety of types, or (b) the same variable can assume values of different types at different times, or (c) a method name can denote more than one *method procedure.* The SOM system reflects the third definition (for example, when a SOM class *overrides* a *parent class* definition of a method to change its behavior). The term literally means "having many forms."

principal The user on whose behalf a particular (remote) *method* call is being performed.

procedure A small section of code that executes a limited, well-understood task when called from another program. In SOM, a *method procedure* is often referred to as a procedure. See also *method procedure.*

process A series of instructions (a program or part of a program) that a computer executes in a multitasking environment.

proxy object In DSOM, a SOM *object* in the *client*'s address space that represents a remote *object.* The proxy object has the same *interface* as the remote object, but each *method* invoked on the proxy is *overridden* by a *dispatch method* that forwards the invocation request to the remote object. Under DSOM, a proxy object is created dynamically and automatically in the client whenever a remote method returns a pointer to an object that happens to be remote.

readers and writers
In the Replication Framework, different processes can access the same replicated object in different modes. A "reader" is a process that does not intend to update the object, but wants to continually watch the object as other processes update it. A "writer" is a process that wants to update the object, as well as continually watch the updates performed by others.

receiver See *target object.*

redispatch stub
A *procedure*, corresponding to a particular *method*, which has the same *signature* as the method's procedure but which invokes somDispatch to dispatch the method. The somOverrideMtab method can be used to replace the procedure pointers in a *class*'s *method table* with the corresponding redispatch stubs. This is done when *overriding* somDispatch to customize *method resolution* so that all *static method* invocations will be routed through somDispatch for selection of an appropriate *method procedure*. (*Dynamic methods* have no entries in the method table, so they cannot be supported with redispatch functionality.)

reference data
Application-specific data that a *server* uses to identify or describe an *object* in DSOM. The data, represented by a sequence of up to 1024 bytes, is registered with DSOM when a server creates an *object reference*. A server can later ask DSOM to return the reference data associated with an object reference. See also *object reference.*

replica When an object is replicated among a set of processes (using the Replication Framework), each process is said to have a replica of the object. From the view point of any application model, the replicas together represent a single object.

replicated object
An *object* for which *replicas* (copies) exist. See *replica.*

run-time environment
The data structures, objects, and global variables that are created, maintained, and used by the functions, procedures, and methods in the SOM run-time library.

scope (Or *naming scope.*) That portion of a program within which an identifier name has "visibility" and denotes a unique variable. In SOM, an *IDL source file* forms a scope. An identifier can only be defined once within a scope; identifiers can be redefined within a nested scope. In a .idl file, modules, interface statements, structures, unions, methods, and exceptions form nested scopes.

serializable See *one-copy serializable*.

server (Or *server implementation.*) In DSOM, a *process*, running in a distributed environment, that executes the *implementation* of an *object*. DSOM provides a default server implementation that can dynamically load SOM *class* libraries, create SOM objects, and make those objects accessible to *clients*. Developers can also write application-specific servers for use with DSOM.

server object In DSOM, every *server* has an *object* that defines *methods* for managing objects in that server. These methods include object creation, object destruction, and maintaining mappings between *object references* and the objects they reference. A server object must be an *instance* of the *class* SOMDServer (or one of its *subclasses*). See also *object reference* and *SOMDObject.*

shadowing In the Emitter Framework, a technique that is required when any of the *entry classes* are subclassed. Shadowing causes instances of the new subclass(es) (rather than instances of the original entry classes) to be used as input for building the object graph, without requiring a recompile of emitter framework code. Shadowing is accomplished by using the macro SOM_SubstituteClass.

signature The collection of types associated with a *method* (the type of its return value, if any, as well as the number, order, and type of each of its arguments).

sister class object
A duplicate of a *class object* that is created in order to save a copy of the class's original *method table* before replacing the method table to customize *method resolution.* The sister class object is created so that some original *method procedures* can be called by the replacement method procedures.

Sockets class A class that provides a common communications interface to Distributed SOM, the Replication Framework, and the Event Management Framework. The Sockets class provides the base interfaces (patterned after TCP/IP sockets); the *subclasses* TCPIPSockets, NBSockets, and IPXSockets provide actual implementations for TCP/IP, Netbios, and Netware IPX/SPX, respectively.

SOM Compiler
A tool provided by the SOM Toolkit that takes as input the interface definition file for a class (the .idl file) and produces a set of *binding files* that make it more convenient to implement and use SOM classes.

SOMClass One of the three primitive *class objects* of the SOM run-time environment. SOMClass is the root (meta)class from which all subsequent *metaclasses* are derived. SOMClass defines the essential *behavior* common to all SOM *class objects.*

SOMClassMgr
One of the three primitive *class objects* of the SOM run-time environment. During SOM initialization, a single *instance* (*object*) of SOMClassMgr is created, called SOMClassMgrObject. This object maintains a directory of all SOM classes that exist within the current process, and it assists with dynamic loading and unloading of class libraries.

SOM-derived metaclass See *derived metaclass.*

SOMDObject The *class* that implements the notion of a *CORBA "object reference*" in DSOM. An *instance* of SOMDObject contains information about an object's *server implementation* and *interface,* as well as a user-supplied identifier.

somId A pointer to a number that uniquely represents a zero-terminated string. Such pointers are declared as type somId. In SOM, somId's are used to represent *method* names, *class* names, and so forth.

SOMObject One of the three primitive *class objects* of the SOM run-time environment. SOMObject is the root class for all SOM (sub)classes. SOMObject defines the essential *behavior* common to all SOM *objects.*

SOMOA (SOM object adapter) class
In DSOM, a *class* that dispatches *methods* on a *server*'s *objects*, using the *SOM Compiler* and run-time support. The SOMOA class implements methods defined in the *abstract* BOA class (its *base class*). See also *BOA class.*

somSelf Within *method procedures* in the *implementation* file for a class, a parameter pointing to the *target object* that is an *instance* of the *class* being implemented. It is local to the *method procedure.*

somThis Within *method procedures,* a local variable that points to a data structure containing the *instance variables* introduced by the *class*. If no instance variables are specified in the SOM *IDL source file*, then the somThis assignment statement is commented out by the *SOM Compiler*.

state (of an object)
The data (*attributes, instance variables* and their values) associated with an *object.* See also *behavior.*

static method Any *method* that can be accessed through *offset method resolution*. Any method declared in the IDL specification of a class is a static method. See also *method* and *dynamic method.*

stub procedures
Method procedures in the *implementation template* generated by the *SOM Compiler*. They are procedures whose bodies are largely vacuous, to be filled in by the implementor.

subclass A *class* that inherits *instance methods*, *attributes*, and *instance variables* directly from another class, called the *parent class, base class*, *superclass,* or indirectly from an *ancestor class*. A subclass may also be called a *child class* or *derived class.*

subclassing The process whereby a new *class*, as it is created (or *derived*), inherits *instance methods, attributes,* and *instance variables* from one or more previously defined *ancestor classes.* The immediate *parent class(es)* of a new class must be specified in the class's *interface declaration.* See also *inheritance.*

superclass See *parent class.*

symbol In the Emitter Framework, any of a (standard or user-defined) set of names (such as, className) that are used as placeholders when building a text template to pattern the desired *emitter* output. When a template is emitted, the symbols are replaced with their corresponding values from the emitter's symbol table. Other symbols (such as, classSN) have values that are used by section-emitting methods to identify major sections of the template (which are correspondingly labeled as "classS" or by a user-defined name).

target object (Or *receiver.*) The object responding to a *method* call. The target object is always the first formal parameter of a *method procedure*. For SOM's C-language bindings, the target object is the first argument provided to the method invocation macro, *_methodName*.

usage bindings
The language-specific *binding* files for a *class* that are generated by the *SOM Compiler* for inclusion in client programs using the class.

value logging In the Replication Framework, a technique for maintaining consistency among *replicas* of a replicated object, whereby the new value of the object is distributed after the execution of a method that updates the object.

view–data paradigm
A Replication Framework construct similar to the Model-View-Controller paradigm in SmallTalk. The "view" object contains only presentation-specific information, while the "data" object contains the *state* of the application. The "view" and "data" are connected by means of an "observation" protocol that lets the "view" be notified whenever the "data" changes.

writers See *readers and writers.*

Index

A

B

C

D

I

S

T