Direct to SOM

Contents

General Description

MrCpp and SCpp support Direct-To-SOM programming in C++. You can write MacSOM based classes directly using C++, that is, without using the IDL language or the IDL compiler.

To use the compiler's Direct-To-SOM feature, combine the replacement SOMObjects headers (described below) from CIncludes, and the MrCpp/SCpp tools from the Tools found in the folder Past&Future:PreRelease:Direct SOMObjects for Mac OS, with an MPW installation that already has SOM 2.0.8 (or greater) installed. Look in the SOMExamples folder for build scripts called DTS.build.script .

The `-som` command line option enables Direct-To-SOM support. When this flag is specified, classes which are derived (directly or indirectly) from the special class named SOMObject will be processed differently than ordinary C++ classes—for these classes, the compiler will generate the MacSOM enabling class meta-data instead of standard C++ vtables. Also when `-som` is specified on the command line, the preprocessor symbol `__SOM_ENABLED__` is defined as 1.

MrCpp and SCpp ship with new, replacement MacSOM header files. The header files have been upgraded to support Direct-To-SOM. Two new header files are of special interest:

Header Requirements

MrCpp and SCpp ship with new, replacement MacSOM header files. The header files have been upgraded to support Direct-To-SOM. Two new header files are of special interest:

• somobj.hh	Defines the root MacSOM class SOMObject. It should be included when subclassing from SOMObject. If you are converting from IDL with C++ to Direct-To-SOM C++, then this file can be thought of as a replacement for both somobj.idl and somobj.xh .
• somcls.hh	Defines the root MacSOM meta-class SOMClass. It should be included when subclassing from SOMClass. If you are converting from IDL with C++ to Direct-To-SOM C++, then this file can be thought of as a replacement for both somcls.idl and somcls.xh .

Several other header files are worth documenting in relation to their usage with Direct-To-SOM.

• som.xh	This standard MacSOM header file defines the procedural interface to SOMObjects™ for MacOS runtime kernel. It is not needed for basic Direct-To-SOM use with the compilers; however, it can be included should you wish to invoke procedural kernel interfaces.
• somdts.h	New for Direct-To-SOM support. Do not include directly from your source. This header file is included internally by the MacSOM header files as needed.
• somobj.xh	Use somobj.hh instead.
• somcls.xh	Use somcls.hh instead.

A new version of all the SOMObjects™ for MacOS header files is provided for use with Direct-To-SOM. These header files replace the older versions, and still allow non-Direct-To-SOM IDL, C and C++ MacSOM class library development.

Note, as part of the normal SOM installation you should also have “somlib” in the MPW {SharedLibraries} and “SOMobjects™ for Mac OS” in your Extensions folder.

Language Restrictions

Direct-To-SOM supports the development and use of MacSOM classes as well as DSOM and CORBA compatible classes. MrCpp and SCpp handle both SOM and DSOM classes uniformly.

SOMObjects and the CORBA model of programming requires a higher degree of encapsulation than is allowed in standard C++. These requirements result in the following language restrictions which apply to either instance objects of or classes derived from SOMObject.

With one exception, the C++ language restrictions listed below apply only to operations on SOM classes or to operations on their instance objects. The exception is that enums must be int (full) sized in the entire compilation unit. Standard C++ (non-MacSOM based) classes can still be declared and used when the `-som` flag is on, and, of course, these restrictions do not apply to them.

(a) When subclassing, the compiler uses the class derivation to determine if the new class should be a MacSOM class. If it finds SOMObject as the base class then the new sub-class becomes a MacSOM class, otherwise not. When subclassing with more than one directly specified parent (i.e., multiple inheritance), all parents must either be MacSOM classes, or none must be MacSOM classes — a class may not inherit from both a MacSOM and non-MacSOM class.

(b) Struct and union MacSOM-based classes are not allowed.

(c) All class inheritance must be <u>virtual</u>.

(d) All data members must be <u>private</u>.

(e) Non-inlined member functions must be unique <u>independent of casing and signature</u>. Thus, in general, function and operator overloading and overriding are not allowed even if the function name has different casing. Because the CORBA standard requires case insensitivity, it is allowed to override a virtual function with the same (case independent) name and signature.

(f) There must be at least one introduced, overridden, or virtual inline member function in the class. The lexically first such function in the class is treated as the “key” member function used to detect whether the class is implemented in the compilation unit.

(g) Inlined member functions have their access restricted to the attributes of the section in which they are defined. Thus public inlines can access only public members, protected inlines can access protected and public members, and private inlines have full access.

(h) No static members (data or functions) are permitted.

(i) Only parameterless constructors (ctors) are allowed.

(j) No copy constructors are allowed. Thus passing MacSOM objects by value and other direct copy assignments are not allowed.

(k) No global MacSOM objects are permitted.

(l) Sizeof() expressions involving SOMObjects and their classes are not allowed.

(m) All enums are int-sized. Specifying `-som` on the command line will imply `-enum max` which will cause all enums to be int sized. This affects the entire compilation unit. (The compiler does not allow multiple sizes of enums in SOM environments.)

(n) Method invocation with explicitly scoped by classname are treated with <u>protected</u> access and the specific classname must be a direct parent of the implementation class. “Scoped” here means that the access specifies the scope explicitly, e.g., `A::member`. Thus, method invocation syntax using classname qualification is used in MacSOM method implementations to perform MacSOM parent call through.

(o) Templates that expand to MacSOM classes are not allowed.

(p) MacSOM-based classes may not have nested class definitions.

(q) Long double member function parameters and return type are not allowed.

(r) Aggregate parameters cannot be passed by value.

(s) Members with a variable number of argument (i.e., “...”) are not allowed.

(t) Only the basic forms of operator <u>new</u> and <u>delete</u> are allowed (e.g., `new(T)`, `delete p`). In other words the placement and array forms of new (e.g., `new (address) T`, `new T[n]`) and array form of delete (e.g., `delete [] p`) are not allowed.

(u) Arrays of MacSOM objects are not allowed.

(v) Embedded MacSOM objects are not allowed, i.e., MacSOM objects declared within other classes.

(w) MacSOM classes are always defined as if they were surrounded by an #pragma align=power and #pragma align=reset. In other words, all MacSOM classes are power aligned.

MacSOM Pragmas

MrCpp and SCpp support seven MacSOM-specific pragmas:

```
#pragma SOMReleaseOrder (method1, method2, ..., methodn)
#pragma SOMClassVersion (className, majorVersion, minorVersion)
#pragma SOMMetaClass (className, metaClassName)
#pragma SOMCallStyle [O]IDL
#pragma SOMModuleName id1::id2::...::idn
```

```
#pragma SOMCheckEnvironment on | off | reset
#pragma SOMCallOptimization on | off | reset
```

These pragmas supply the compiler with information it needs to provide to the MacSOM runtime kernel or for the compilation itself. Please refer to the SOMObjects Developers Toolkit documentation, specifically, the Users Guide, for more information regarding release order, class version, meta class programming and call styles. The SOMCheckEnvironment and SOMCallOptimization pragmas are specific to MrCpp and SCpp and are fully described here.

The syntax for these is compatible with other Direct-To-SOM C++ compilers. All these pragmas except for SOMCheckEnvironment and SOMCallOptimization may only occur within the scope of the class definition for which they are intended. The pragmas may occur more than once within the class but only if they specify exactly the same information. An error is reported if they are inconsistent.

`#pragma SOMReleaseOrder (`$\texttt{\textbf{method}}_1$**`, `**$\texttt{\textbf{method}}_2$**`, ..., `**$\texttt{\textbf{method}}_n$**`)`**

As with IDL, MacSOM based classes must specify the release order of the member functions of the class. This is done using the SOMReleaseOrder pragma. The $\texttt{method}_i$’s in the pragma are simple member function <u>(case independent</u>) method names with no qualification and no signature.

The SOMReleaseOrder pragma must specify every member introduced (i.e., no overrides) by the class. Once the release order is specified, <u>and</u> the class made available to clients, that order must not be changed. If a member is deleted, its name must remain in the release order. If a new member is added, its name should be added at the end of the release order list. If a member is migrated up in the ancestry, its name will appear in both the ancestor and also in its original release order.

If the SOMReleaseOrder pragma is omitted, the assumed release order will be the lexical order that the member functions appear in the class. This is permitted since it can be a inconvenient to maintain the pragma during initial class development. But the pragma should be provided when the class is released for use by clients. If the pragma is supplied, it is considered an error condition to not list all the class’ members.

`#pragma SOMClassVersion (className, majorVersion, minorVersion)`

The SOMClassVersion pragma specifies the version numbers for the MacSOM class. If the pragma isn’t provided, zeros are assumed. Version numbers must be non-negative. If the class is being defined, then its version numbers are passed to the MacSOM kernel in the class meta-data. When an instance of the class is instantiated via the new operator, the version numbers are passed to the runtime kernel which performs a consistency check to make sure the class implementation is not out of date.

`#pragma SOMMetaClass (className, metaClassName)`

A class that defines the implementation on <u>class</u> objects is called a <u>metaclass</u>. Just as an instance of a class is an object, so an instance of a metaclass is a class object. Moreover, just as an ordinary class defines methods that its objects respond to, so a metaclass defines methods that a class object responds to.

SOMClass is the root class for all SOM metaclasses. SOMClass itself is a descendent of SOMObject and therefore inherits all the generic object methods; this is why instances of a metaclass are class objects (rather than simply classes) in the MacSOM runtime. All metaclasses must be descendants, directly or indirectly, of SOMClass.

The default metaclass for a MacSOM class is SOMClass. The SOMMetaClass pragma allows the user to pick another metaclass. It is an error if the specified `metaClassName` does not have SOMClass as one of its ancestors. Also a class cannot be defined as its own metaclass. Thus the `className` and `metaClassName` parameters must never specify the same class.

`#pragma SOMCallStyle OIDL`

MacSOM itself supports two call styles, an older style that does not support DSOM, called OIDL, and the newer that does, called the IDL call style. MrCpp and SCpp, by default, assume that classes defined using Direct-To-SOM use the newer IDL call style. When using this call style, all methods must have an Environment pointer parameter as the first parameter. Just as when using MacSOM without the DTSOM compiler support, the environment parameter is used to communicate exception information following method invocation. This environment parameter is explicitly required in the (DTSOM) C++ method specifications. The pragma for OIDL is supplied and used by the SOM base classes SOMObject and SOMClass. Note that when overriding methods declared in SOMObject or SOMClass, the override method declaration should appear exactly the same as the method when originally introduced. That is, for SOMObject and SOMClass introduced methods, no environment parameter is used; however, for other classes an environment parameter is required.

#pragma SOMModuleName $\mathtt{id_1::id_2::...::id_n}$

When an instance of a SOM object is created, that class' name is made known to the SOM runtime since the name is generated as part of the static data associated with any MacSOM object. This means that there is the possibly of name collision between two SOM objects (usually provided from two different suppliers). The SOMModuleName pragma should be used to avoid this problem. It approximates the module name functionality in IDL.

The $\mathtt{id_n}$'s in the SOMModuleName pragma specify simple identifiers. Any number of identifiers may be specified, each separated by a '`::`'. The sequence of identifiers is used to qualify all the externally visible names associated with a MacSOM object. In other words, the token table name and the class name generated as part of the class' static data so that the class is unique with respect to the SOM runtime environment. For example, for class `X`, the token table name, `XClassData` becomes

id_1_id_2_..._id_n_XClassData. The class name that will be known to the SOM runtime becomes id_1::id_2::...::id_n::X.

```
#pragma SOMCheckEnvironment on | off | reset
```

As discussed previously, the compilers assume the IDL call style by default. Thus all introduced members of all descendants of SOMObject and SOMClass have an Environment pointer parameter as the first parameter. The Environment is a data structure that contains environmental information and is also used to return exception data to a client. After a call to an IDL introduced member returns, the caller can look at the `_major` field in the Environment data. If the value of `_major` is not equal to `NO_EXCEPTION` (0), there was an exception returned by the call. The caller can retrieve the exception name and value using the somExceptionId and somExceptionValue routines.

Assume the analysis of the exception is not done at the call site but rather in a routine called `__som_check_ev(Environment *)`. Then a typical member call might look like,

```
   member(&ev, other args...);
   __som_check_ev(&ev);
```

This can get tedious to do on every member call, so the SOMCheckEnvironment pragma is provided to tell the compiler to automatically insert a call to `__som_check_ev` which should check `_major` and act accordingly if it is non-zero. `__som_check_ev` is written by the user and must have the following prototype (which is defined in somdts.h),

```
   extern "C" void __som_check_ev(Environment *);
```

Note that `__som_check_ev` should clear the error status of the Environment (by calling `somExceptionFree`), otherwise the next SOM call that returns will see the same error again!

In addition to inserting a check after each member call, when SOMCheckEnvironment is on, the compiler will insert a call to `__som_check_new` after each operator new call.

```
   T *p = new T;
   __som_check_new(p);
```

The user also supplies `__som_new_new` which should check to see if the allocation succeeded. It has the prototype (defined in somdts.h),

```
   extern "C" void __som_check_new(SOMObject *);
```

These checks are inserted by the compiler as long as SOMCheckEnvironment is on. If they are not needed, `#pragma SOMCheckEnvironment off` may be specified. This is also the default setting.

Finally, a `reset` option is provided in case nesting of this pragma is needed. It restores the SOMCheckEnvironment state to what it was at the time of the most recent corresponding `on`.

```
#pragma SOMCallOptimization on | off | reset
```

Inserting the additional check code enabled by the SOMCheckEnvironment pragma will obviously increases code size. Even without the checks, just doing a member call requires accessing a pointer in the SOM data (generated by the compiler) and indirectly jumping through that pointer. On the PowerPC, a size optimization is available to minimize the call site code down to a single instruction (not counting the parameter setup)! Unfortunately, for complex reasons related to parameter-passing models, this optimization is not available on the 68K (SCpp). So the following discussion applies only to MrCpp.

The size optimization can be enabled by using `#pragma SOMCallOptimization on`. The optimization involves moving most of the member call code to a small code sequences referred to as “glue” code. The glue code is generated as part of the compilation unit. There is one or two glue code routines for each explicitly called member (one unless the same member is called with both SOMCheckEnvironment on and off). But all calls to the same member go through the same glue code associated with that member. The member call becomes a single instruction to the glue routine (ignoring parameter setup). The glue is defined as if a `#pragma internal` was done so there is no `NOP` following the call.

Each glue routine is responsible for determining the member pointer only. All the calling (and the requisite `NOP` following the call) and Environment or `NULL` checking is constant and therefore factored out into a small set of library routines. The glue code therefore branches to these library routines. These routines are located in PPCRuntime.o.

In an experimental implementation of this optimization in OpenDoc 1.1 code size was reduced by approximately 10%.