MrC[pp] Specific #Pragmas

The Additional #pragmas added to MrC[pp]

Ira L. Ruben & Fred Forsman
Apple Computer, Inc.

November 8, 1998
Revision 3.4

Table Of Contents

Revision History

Revision	Date	Comments
1.0	?	Original paper listing code positioning pragmas only
2.0	?	CFM pragmas added.
2.1	2/16/96	‘reset’ option added to CFM pragmas
3.0	4/16/96	Reorganized and made this document a repository for all pragmas added to MrC[pp]. Therefore, the traceback, unused, and DTSOM pragmas were incorporated into this document. The document was “formalized” with a TOC and this revision history.
3.1	4/24/96	Added options pragmas and removed the single quotes around all keywords.
3.2	10/9/96	New SOM pragmas - SOMModuleName, SOMCallOptimization, SOMCheckEnvironment. New pragmas - disjoint, ignore.
3.3	11/15/96	precompiler_target and dump pragmas added.
3.4	11/8/98	AltiVec pragmas added.

1.0 New Pragmas For MrC[pp] -- Overview

This document describes the new pragmas that have been added to MrC[pp]. They are divided into the following major categories):

Code Positioning Pragmas

- `#pragma [no]inline_func [list] <func_list>`
- `#pragma [no]inline_site [list] <func_list>`
- `#pragma seldom`
- `#pragma outofline`
- `#pragma segment <segname> [[list] <func_list>]`

CFM Pragmas

- `#pragma export on | off | reset | [list] <name_list>`
- `#pragma import on | off | reset | [list] <name_list>`
- `#pragma internal on | off | reset | [list] <name_list>`

Direct-To-SOM Pragmas

- `#pragma SOMReleaseOrder (method`$_1$`, method`$_2$`, ..., method`$_n$`)`
- `#pragma SOMClassVersion (className, majorVersion, minorVersion)`
- `#pragma MetaClass (className, metaClassName)`
- `#pragma CallStyle [O]IDL`
- `#pragma SOMModuleName id`$_1$`::id`$_2$`::...::id`$_n$
- `#pragma SOMCheckEnvironment on | off | reset`
- `#pragma SOMCallOptimization on | off | reset`

AltiVec™ Pragmas

- `#pragma altivec_model on | off | reset`
- `#pragma altivec_codegen on | off | reset`
- `#pragma altivec_vrsave on | off | reset | allon`

Option Pragmas

- `#pragma options align [=] mac68k | power | byte | packed | reset`
- `#pragma options inline [=] on | all | none | off | 0 | 1 | 2 | 3 | 4 | 5 | reset`
- `#pragma options opt [=] off | none | local | size | speed[,<modifiers>] | reset`

Miscellaneous Pragmas

- `#pragma traceback [list] <name_list>`
- `#pragma unused [list] <name_list>`
- `#pragma ignore id,...`
- `#pragma disjoint [list] <disjoint-list>`

The code positioning pragmas give the user more control over the positioning of code generated by the compiler or placed by the linker. The CFM pragmas are the CFM 68K counterparts to

---

™ AltiVec is a registered trademark of Motorola, Inc.

allow for more efficient code generation for the Macintosh CFM DLL model. The Direct-To-SOM are, of course, needed for MacSOM support. The option pragma allow for overriding selected command line options. Finally, the miscellaneous category is for various other pragmas that don't fall into the other categories.

1.1 General Syntax for the 'list' forms

Many of the MrC[pp] pragmas have a "list" form, e.g.,

```
#pragma [no]inline_func [list] <func_list>
```

The keyword `list` is always optional unless no other `<func_list>` follows it. The `<func_list>` takes two forms,

```
<func_list> ::= <name_list> | ( <name_list> )
```

In other words, a `<func_list>` may optionally be enclosed in parentheses[1], where `<name_list>` is a list of function names of the form,

```
<name_list> ::= <func> | <name_list> , <func>
<func>      ::= [::]<id> | <member> | <ctor> | <dtor> |
                <operator> | <template>                    (only <id> for C)

<member>    ::= <id> :: <id> | <member> :: <id>
<ctor>      ::= <id> :: <id>    (<id>s are the same)
<dtor>      ::= <id> :: ~<id>                          (<id>'s are the same)
<operator>  ::= operator <op>
<template>  ::= <id> < <template_args> > :: <id> |
                <id> < <template_args> > :: <member>
```

C++ names can be a simple id (as in the C case), or a member name. Member names, in turn, can take to form of ctors (e.g., A::A), dtors (e.g., A::~A), simple members (e.g., A::foo), operators (e.g., A::operator<< or B::operator int*), and template members (e.g., T<char, 2>::foo).

If the *first* member of the `<name_list>` is `list::<id>`, then an ambiguity exists in determining whether to ignore 'list' as a "noise" word. MrC[pp] elects to treat 'list' as "noise" in that case. If `list::<id>` must be specified, then it cannot be specified as the first member or the optional parentheses. may be used.

Depending on the specific pragma, there may be further restrictions placed on the syntax and/or semantics. For example, only simple function names, no C++ member names, or a name cannot be referenced or defined. Such restrictions where appropriate in the descriptions of the individual pragmas.

2.0 Code Positioning Pragmas

The code positioning pragmas give the user more control over the positioning of code generated by the compiler or placed by the linker. They are grouped into three general categories:

- Inlining pragmas

---

1 The `SOMReleaseOrder` pragma is shown using the parenthestical form. That's done because it is the "standard" syntax for that pragma in order to be compatible with other diret-to-SOM compilers.

- Code block placement pragmas
- Segmentation

2.1 Inlining Pragmas

The [no]inline_func and [no]inline_site pragmas control inlining of function bodies. You have the choice of inlining all selected bodies or just inlining them for specific calls.

2.1.1 #pragma [no]inline -- Function body inlining of all calls

The inline_func pragma specifies functions that are to be candidates for inlining are all places in the source following the pragma. Conversely, the noinline_pragma specifies that its functions are not to be inlined even if explicitly specified for inlining using the inline keyword in C++.

Syntax **`#pragma [no]inline_func [list] <func_list>`**

See section 1.1 for a description of the list form.

Semantics

(1) Any function specified on the function list must not be defined or called. Declarations, however, *are* permitted.

Example 1:

```
class T {
   public:
      member1();
      member2() {}
      member3();
      member3(int);
};

void foo();                    // an explicit declaration

void T::member1()
{
   bar();                      // an implicit declaration and call
}

#pragma inline_func foo, bar, T::member1, T::member2, T::member3
```

In the example pragma the valid function references are foo and T::member3. None of these functions are defined, only declared. On the other hand the references to T::member1 and T::member2 are invalid since these are defined. The reference to bar is also invalid since it was called.

Note, invalid references and syntax errors are reported as errors (as opposed to warnings). References to existing function (member) declarations are validated that the reference truly is a function.

n In the early design of the [no]inline_func pragmas it was decided that all references to any of the functions were to be considered an error. This requires that the pragma be positioned even before declarations of functions referenced by the pragma. It also means no member functions could be specified if the class definition existed prior to the pragma. From an

implementation point of view the pragma reports a “reference” error for a function if the function or it’s class are “seen” in the symbol table.

This is considered overly restrictive. As long as there are no definitions (calls are restricted in either design) to the functions, it is sufficient to allow them to be specified by the pragmas. It also means that the pragma can be placed with the implementation rather than the (header) declarations. Further, it is more consistent with the requirement that template definitions exists (but not their instances) prior to specifying an instance function in the pragma. There is no way to support templates without this requirement. o

(2) Template references require that the template definition exist prior to the pragma. That's the only way the references to instances can be parsed. However, the specified instances must *not* exist.

Example 2:

```
template <class T, int N> class TEMP {
   public:
      TEMP(T i) {i = N;}
      T foo(T x) {return x+i+N;}
      int i;
};

template <class T> T TempFunc(T x)
{
   return x;
}

TEMP<int, 1> g;

#pragma inline_func TEMP<int,1>::foo, TEMP<int,2>::foo, TempFunc

void placeForTemplates()
{
   TEMP<int, 2> x(1);
   char c = TempFunc('x');
}
```

The pragma reference to TEMP<int,1> is invalid since that instance is already defined (g). The TEMP<int,2> is legal since the instances are not referred to until the procedure that follows the pragma. The reference to TempFunc is similarly valid for the same reason. It shows a template function. These are referred to exactly like any other function.

(3) A valid reference to a function implies that function is a candidate for inlining wherever it is used.

Example 3:

```
void foo();

#pragma inline_func foo

void bar()
{
   foo();
}
```

```
void foo() {}

void main()
{
   foo();
}
```

foo() is a candidate for inlining in both bar() and main(). Note the prototype for foo() is required in C++ and, as discussed in (1) above, may be either before or after the pragma.

(4) A reference to a function in C++ implies a reference to all of its overloads, if any. In example 1, the reference to T::member3 is a reference to both T::member3() and T::member3(int).

(5) Any valid function specified in the [no]inline_func pragma that has not been defined by the end of the compilation unit will be reported as "undefined" warning.

(6) Duplicate references to the same functions are reported as a warnings. References to functions specified for inline_func are reported as an error if an attempt is made to define them for noinline_func and vice versa.

(7) If there is a resolution conflict between the inline_func pragma and the auto-inliner, then the pragma will override the analysis of the auto-inliner.

(8) Turning inlining or optimization off from the command line overrides these pragmas.

2.1.2 #pragma [no]inline_site -- Function body inlining of specific calls

The inline_site pragma specifies that its specified functions are to be inlined only within the blocks that it is placed. The noinline_site pragma indicates that the specified functions are not to be inlined in the containing block.

Note the difference between [no]inline_func and [no]inline_site is that [no]inline_func basically refers to function definitions while [no]inline_site refers to specific call sites.

Syntax **`#pragma [no]inline_site [list] <func_list>`**

The syntax for the [no]inline_site pragma is identical to inline_func described above.

Semantics

(1) The [no]inline_site pragma may only be used *within* a function (brace-enclosed block) and applies only to that block and all blocks nested within it.

(2) The [no]inline_site pragma may *not* be used in template (member) function definitions. Template definitions can be viewed basically as macro definitions. As such, preprocessor statements like #pragma are processed as seen. If it occurs within a template definition it is not in a function. Thus statement (1) applies.

(3) Inner blocks with their own [no]inline_site pragmas override enclosing block [no]inline_site pragmas.

(4) Duplicate references to the same functions is reported as a warning. References to functions specified for inline_site are reported as an error if an attempt is made to define

them for noinline_site and vice versa *in the same block.*

(5) References to defined or declared functions are validated (e.g., they must actually be functions). References to yet-to-be-defined functions are also permitted but obviously these are not validated.

(6) A reference to a function in C++ implies a reference to all of its overloads.

(7) If a [no]inline_site pragma appears in the middle of a block, then only functions in the block from the point at which the pragma occurs are candidates for [no]inlining. The pragma has no effect on calls in the block before the pragma.

(8) The [no]inline_site pragmas have precedence over the [no]inline_func pragmas for the same functions.

(9) Any valid function specified in the [no]inline_site pragma that has not been called by the end of the block will be reported as "unreferenced" warning (at the end of the block containing the pragma).

Example:

```
#pragma inline_func foo

void bar(int i, int j, int k)
{
   if (i == 1) {
      x();                              // not explicitly inlined
      #pragma inline_site x, y, z
      x();                              // candidate for inlining
      if (j == 2) {
         y();                           // candidate for inlining
         if (k == 3) {
            #pragma noinline_site y, foo
            y();                        // not inlined
            foo();                      // not inlined
         } // 3
      } // 2                            // unreferenced warning for z
   } // 1
}
```

In the above example, the first x() is not a candidate for inlining (assume this a C example) since it occurs before the inline_site pragma. The y() in the j==2 block is a candidate for inlining but not in the k==3 block since a more local noinline_site pragma overrides the outer one. Similarly, the noinline_site pragma overrides the inline_func for foo. At the end of the j==2 block a warning will be issued for z since it was not called.

(9) If there is a resolution conflict between the inline_site pragma and the auto-inliner, then the pragma will override the analysis of the auto-inliner.

(10) Turning inlining or optimization off from the command line overrides these pragmas.

2.2 Code Block Placement Pragmas

The seldom and outofline pragmas control movement of the blocks of code containing these pragmas.

2.2.1 #pragma seldom | outofline -- Move code locks based on anticipated use

Seldom specifies that a block (of code) is seldom executed and that the compiler may move the block to the end of its function (CSect).

Outofline implies that a block is rarely executed and that the compiler may move the block out of the current function (CSect). The compiler will try to generate code for the block for placement in a separate unique Csect. The PPC Linker, in turn, can than collect these Csects and place it at the end of the program.

Syntax **`#pragma seldom`**
`#pragma outofline`

Semantics

(1) The pragmas may only be used *within* a function (brace-enclosed block) and applies only to that block and all blocks nested within it (i.e., these have the same scope rules as #pragma [no]inline_site).

(2) The seldom and outofline pragmas may *not* be used in template (member) function definitions. Template definitions can be viewed basically as macro definitions. As such, preprocessor statements like #pragma are processed as seen. If it occurs within a template definition it is not in a function. Thus statement (1) applies.

(3) The outofline pragma has precedence over seldom. Thus an outofline pragma in an outer block will override any seldom pragmas in a blocks nested within it.

Example 1:

```
- - -
{
   #pragma outofline
   - - -
   {
      #pragma seldom
      - - -
   }
}
- - -
```

The seldom pragma in the inner block has no effect. No warning or error is reported for this situation.

(4) If both a seldom and outofline pragmas appear in the same block, a warning is issued and the more recent pragma applies to that block.

(5) The pragmas apply to the entire block that contains them even if the pragma does not appear at the beginning of the block.

(6) Speed optimizations like inlining may be turned off if they do not lead to smaller code size.

Example 2:

```
int foo(int i)
{
   switch (i) {
      case 1:   f(i);
                break;
      case 2: { #pragma seldom
                warning(1);
              }
      default:{ #pragma outofline
                fatal_error(999);
              }
   }
}
```

In the above example, case 1 is "normal" code. Case 2 is a warning condition that is rare but possible so it is to be placed at the end of the function. The default case is a "this cannot happen" case and is placed in a separate Csect at the end of the program. Note that braces are required to indicate a (nested) block. It is not sufficient to just place these pragmas in the switch cases. That would affect the entire switch "block".

(7) Turning optimization off from the command line overrides these pragmas.

2.3 Segmentation

The segment macro provides for grouping of entire functions into "segments" identified by segment names.

2.3.1 #pragma segment -- Collect functions into groups

The segment macro provides for grouping of entire functions into "segments" identified by segment names.

Syntax **`#pragma segment <seg_name> [[list] <func_list>]`**

```
<seg_name>  ::= <identifier> | <string>
<func_list> ::= see #pragma [no]inline_func for syntax
```

Segment names may take the form of a single identifier or a double quoted (possibly concatenated) character string. Case is significant.

The syntax for the `<func_list>` is identical to that described for [no]inline_func.

Semantics

(1) This pragma may only appear *outside* of any function definitions.

(2) Any function on the list must not be defined or called. See #pragma [no]inline_func for the complete semantics on the functions in the list since the segment functions have identical semantics ([no]inline_funct semantics items 1 and 2).

(3) A reference to a function in C++ implies a reference to all of its overloads.

(4) For the list form, any valid function specified in the segment pragma that has not been defined by the end of the compilation unit will be reported as “undefined” warning.

(5) Duplicate references to the same functions are reported as a warnings.

(6) The non-list form applies to the next function in the compilation unit and all following functions unless changed by another non-list segment pragma.

(7) The list form defines the segments for specific functions and takes precedence over the non-list form.

(8) If the string is defined as null (""), the following functions are not defined for any segment. It is error to specify a null string with the list form of this pragma.

(9) All function defined for a particular segment are collected together at link time to be placed in a single common segment (should that be a common Csect here?)

Example:

```
#pragma segment Seg1
void a() {...}                // a in Seg1

#define TWO "2"
#pragma segment "Seg" TWO
void b() {...}                // b in Seg2

#pragma segment Seg3 x, y, z

void c() {...}                // c in Seg2
void x() {...}                // x in Seg3
void d() {...}                // d in Seg2
void y() {...}                // y in Seg3
void e() {...}                // e in Seg2
void z() {...}                // z in Set3

#pragma segment ""
void f() {...}                // not in any explicit segment

#pragma segment "" u, v       // error -- null not allowed for lists
```

3.0 CFM Pragmas

The CFM pragmas allow for proper and efficient code generation for the Macintosh “CFM” DLL model. There are several goals for these pragmas:

- Support the code fragment programming model.
- Allow efficient code generation.
- Preserve language semantics when the pragmas are not used.

All the CFM pragmas have the same following general syntax and common semantics.

Syntax **#pragma <cfm_pragma> <cfm_pragma_option>**

```
<cfm_pragma>        ::= export | import | internal
<cfm_pragma_option> ::= on | off | [list] <cfm_name_list> | reset
<cfm_name_list>     ::= [(] <the_names> [)]
<the_names>         ::= <name> | <the_names> , <name>
<name>              ::= <func> | <id>
```

The are two forms for these pragmas; the on/off modal form and the list form. In other words,

```
#pragma <cfm_pragma> on
#pragma <cfm_pragma> off
#pragma <cfm_pragma> reset
#pragma <cfm_pragma> [list] <name_list>
```

The list form specifies a list of variable or function names. The syntax for function names is identical to that described for `<func>`'s which make up the `<func_list>`'s described for the [no]online_func pragmas. The syntax for a `<cfm_name_list>` is the same as a `<func_list>` with respect to the treatment of the keyword 'list' and parentheses.

Common CFM Pragma Semantics

(1) This pragma may only appear *outside* of any function definitions.

(2) Template references require that the template definition exist prior to the pragma.

(3) The list form of these pragmas may only specify variables before they are defined or referenced. Thus only externs are acceptable *declared* variables to the list form.

(4) The list form of these pragmas may only specify functions before they are defined or called. Thus functions must be declared (explicitly through an extern or are class member functions).

(5) A reference to a member function in C++ implies a reference to all of its overloads. References to C++ class variables is not allowed.

(6) Variables and functions referenced by the list forms must be "known" prior to the pragma. They can be defined or declared. What specifically is legal depends on the pragma.

(7) These pragmas are never applied to file scoped symbols, i.e., those using the keyword "static". Specifying such a symbol in the list form will be reported as an error by the pragma.

(8) Subject to the specific semantics of the CFM pragma (described later), an 'on' CFM pragma affects all functions and variables following the pragma up to the matching 'off'.

(9) Subject to the specific semantics of the pragmas, "inner" nested 'on's override or merge with "outer" nested 'on's.

(10) The 'reset' option is similar to the 'off' option except that the state is reset to what it was at the time of the most recent corresponding 'on'.

3.1 #pragma export - Mark symbols as exported from the compilation unit

Syntax
```
#pragma export on
#pragma export off
#pragma export reset
#pragma export [list] <name_list>
```

Semantics

The export pragma has no effect on generated code or data. Its sole purpose is to mark symbols as exports in the generated object file (and thus in the final executable code fragment).

Note, if developers wish to have automatic generation of exports by the compiler and linker, they should use the export pragma in their implementation files.

(1) The export pragma is never applied to file scoped symbols, i.e., those using the keyword "static". Specifying such a symbol in the list form will be reported as an error by the pragma.

(2) The export and import pragmas are orthogonal and may be applied independently.

(3) The export and internal pragmas are orthogonal for variables and may be applied independently.

(4) Specifying export and internal for functions is an error since these pragmas conflict with respect to functions (export says generate a TVector while internal says don't).

3.2 #pragma import - Specify symbols to be imported from another compilation unit

Syntax
```
#pragma import on
#pragma import off
#pragma import reset
#pragma import [list] <name_list>
```

Semantics

The import pragma indicates that a symbol is to be treated as though it were imported from another fragment. This affects code generation for references to both variables and functions. Imported variables are addressed indirectly. Imported functions are called using cross fragment "glue". This includes calls within the same source file (and thus includes recursive calls).

Developers of shared libraries are expected to use the same header file for both external clients and their own internal builds. They should use the import pragma in this header. Clients obviously need to treat these symbols as imports and internal builds need to treat them as imports if they are to be updatable/patchable. Note that this says the library exports should be tagged import in the public header. This is consistent with the client's view.

(1) The import pragma is never applied to file scoped symbols, i.e., those using the keyword "static". Specifying such a symbol in the list form will be reported as an error by the pragma.

(2) The import and export pragmas are orthogonal and may be applied independently.

(3) The import and internal pragmas are mutually exclusive. The import pragma will disable the internal status if it is set.

3.3 #pragma internal - Specify symbols specifically private to the compilation unit

Syntax `#pragma internal on`
`#pragma internal off`
`#pragma internal reset`
`#pragma internal [list] <name_list>`

Semantics

Internal variables may be addressed directly off of the RTOC (i.e., they may be allocated directly "in" the TOC). Internal functions may always use local calling conventions and do not require a function descriptor ("TVector").

The internal pragma should be used in private headers to allow optimal code generation for variables and routines which are referenced from multiple source files and hence cannot use C static scoping.

The internal pragma could also be conditionally applied to variables in the public header for internal builds to get optimal code generation. The internal pragma should never be applied to functions in a public header.

(1) The internal pragma is never applied to file scoped *variables*, i.e., those using the keyword "static". Such variables are implicitly internal. Specifying such a symbol in the list form will be reported as an error by the pragma.

(2) Internal *does* affect file scoped (static) functions indicating that their function descriptor ("TVector") may be omitted.

(3) It is an error to take the address of a "internal" function.

(4) The internal and export pragmas are orthogonal for variables and may be applied independently.

(5) Specifying internal and export for functions is an error since these pragmas conflict with respect to functions (export says generate a TVector while internal says don't).

(6) The internal and import pragmas are mutually exclusive. The internal pragma will disable the import status if it is set.

(7) A command line option (`-imax` *n*) is provided to supply the maximum size for which a internal variable is a candidate for placing directly into the TOC. The default is to not place any variables larger than 512 bytes in the TOC.

> n According to the paper on which these CFM pragmas are based, i.e., "C Compiler Pragmas for Macintosh CFM Runtime", by Alan Lillich and Erik Eidt, there is the following recommendation"
>
> All compilers should support referencing internal variables directly off of RTOC/A5, i.e., allocation of variables "in the TOC". This should be implicit for file scoped variables. There

should be a size threshold, with “small” variables using direct references and “large” variables remaining indirect. This promotes use of direct references and 16 bit offsets as widely as possible without bounding the total internal data at 64KB. The size threshold should be settable as a command line option.

This implies that all internal variables up to a command line settable size should be placed directly in the TOC. A suggested default size is around 1K. We have chosen 512. o

4.0 Direct-To-SOM Pragmas

The Direct-To-SOM pragmas are only valid when the `-som` command line option has been specified to enable MacSOM in the MrCpp (i.e., C++ only) compiler. These pragma’s are used to provide MrCpp with information it needs to provide to the MacSOM runtime kernel. Please refer to the SOMObjects Developers Toolkit documentation, specifically, the Users Guide, for more information regarding release order, class version, meta class programming and callstyles.

The syntax for these compatible with other Direct-To-SOM C++ compilers. These pragmas 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.)

4.1 #pragma SOMReleaseOrder - Specify class member function release order

Syntax **`#pragma SOMReleaseOrder (method`$_1$`, method`$_2$`, ..., method`$_n$`)`**

Semantics

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 `method`$_i$’s in the pragma are simple member function (*case independent*) 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, *and* 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

4.2 #pragma SOMClassVersion - Specify a class ‘s version

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

Semantics

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.

4.3 #pragma SOMMetaClass - Specify a class 's metaclass

Syntax `#pragma SOMMetaClass (className, metaClassName)`

Semantics

A class that defines the implementation on *class* objects is called a *metaclass*. 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.

4.4 #pragma SOMCallStyle - Specify a class 's member function call style

Syntax `#pragma SOMCallStyle OIDL`

Semantics

MacSOM itself supports two callstyles, an older style that does not support Direct-to-SOM, called OIDL, and the newer that does, called the IDL callstyle. MrCpp by default assumes that classes defined using Direct-To-SOM use the newer IDL callstyle. When using this callstyle, all methods must have an Environment pointer parameter (`Environment *`) as the first parameter. Just as when using MacSOM without the Direct-to-SOM compiler support the environment parameter is used to communicate exception information following method invocation. This environment parameter is explicitly required in the (Direct-to-SOM) 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 and environment parameter is required.

4.5 #pragma SOMModuleName - Specify a class 's module name

Syntax `#pragma SOMModuleName` id_1`::`id_2`::...::`id_n

Semantics

When an instance of a SOM object is created, that class's 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 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's 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`.

4.6 #pragma SOMCheckEnvironment - Control SOM Environment checking

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

Semantics

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`.

4.7 #pragma SOMCallOptimization - Control SOM member call optimization

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

Semantics

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%.

5.0 AltiVec Pragmas

These pragmas are used to control the handling of AltiVec.

5.1 #pragma altivec_model - Control acceptance of the AltiVec model

Syntax **`#pragma altivec_model on | off | reset`**

Semantics

This pragma is used to either temporarily or permanently override accepting the AltiVec

extensions as specified by the command line -vector on or `-altivec_model on`. The setting remains in effect until the next `altivec_model` pragma is encountered.

This pragma may be placed anywhere within the compilation unit. If `reset` is specified, the setting is reset to what was specified or implied by the command line.

5.2 #pragma altivec_codegen - Control AltiVec (vectorization) optimizations

Syntax **`#pragma altivec_codegen on | off | reset`**

Semantics

This pragma is used to either temporarily or permanently override vectorization of code as specified by the command line `-opt size` or `speed altivec_codegen` parameter. When vectorization is enabled, code generation is allowed to take advantage of the AltiVec architecture as a possible optimization.

When used outside of a function, then the pragma overrides the command line until another `#pragma altivec_codegen` is encountered outside of any functions. If `reset` is specified, the setting is reset to `off`.[2]

If the pragma is placed inside a function body (i.e., anywhere between its enclosing braces), then the pragma temporarily overrides the current setting *for that function only*. The setting applies to the entire function no matter where within the function the pragma is placed. If more than one `#pragma altivec_codegen` is placed within the function, then it's an error if they have different settings. The `reset` option is not permitted when the pragma is used within functions. Following the function, the default setting is reset to what was in effect prior to that function.

Note: This pragma is recognized but no implicit AltiVec vectorization optimizations are performed at this time.

5.3 #pragma altivec_vrsave - Control handling of VRsave

Syntax **`#pragma altivec_vrsave on | off | reset`**

Semantics

This pragma is used to either temporarily or permanently override maintaining of the VRsave register as specified by the command line `-vector on,[no]vrsave` or `-altivec_model on,[no]vrsave`. When enabled, function prologs and epilogs have additional code to properly maintain VRsave to indicate which vector registers are currently in use.

When used outside of a function, then the pragma overrides the command line until another `#pragma altivec_vrsave` is encountered outside of any functions. If `reset` is specified, the setting is reset to what was specified or implied by the command line. The `allon` option is not permitted when the pragma is used outside of a function.

---

2 Eventually there may be a command line option, in which case reset will `reset` to the setting specified or implied by the command line.

If the pragma is placed inside a function body (i.e., anywhere between its enclosing braces), then the pragma temporarily overrides the current setting *for that function only*. The setting applies to the entire function no matter where within the function the pragma is placed. If more than one `#pragma altivec_vrsave` is placed within the function, then it's an error if they have different settings. The `reset` option is not permitted when the pragma is used within functions. Following the function, the default setting is reset to what was in effect prior to that function.

It is not recommended that VRsave handling be turned off since interrupt handlers need VRsave in order to know which vector register need to be preserved across interrupts. However there is a price to be paid in prolog/epilog overhead in maintaining VRsave. It is possible to safely turn off VRsave handling if it is known that the VRsave register reflects all possible vector registers that can be in use. Using the `allon` option indicates that the function containing this option will define VRsave as having the value of all ones thus indicating all vector registers are in use. All functions called by this function and their descendants can then be safely set to *not* maintain VRsave. It is the user's responsibility to ensure VRsave is properly controlled in this call chain.

6.0 Option Pragmas

The option pragma allow for overriding selected command line options. The following sections document the MrC[pp] additions to the standard #pragma options statement.

6.1 #pragma options align - Set data structure alignment

Syntax `#pragma options align [=] mac68k | power | byte | packed | reset`

The `'='` is optional, macro substitution is performed, and case is significant.

Semantics

This pragma is used to override the default alignment specified by the command line `-align` option. The syntax for the pragma following the `align` keyword is identical to that of the command line (except for `reset` which is specific to this pragma). The parameters have the same meaning as the command line.

`mac68k`	Use 680x0 alignment.
`power`	Use PowerPC alignment.
`byte`	Use byte (i.e., no) alignment.
`packed`	Identical to `byte`.
`reset`	Return to the previous alignment.

These may be placed anywhere within the compilation unit. Specifying `reset` causes the default alignment to be reset to what was in effect prior to the most recent previous align pragma, or what was implied by the command line if there is no previous align pragma.

.See MrC/MrCpp C/C++ Compiler for the Power Macintosh for further details on this pragma. [That manual should be updated to include the `byte` and `packed` options].

6.2 #pragma options inline - Set inlining level

Syntax `#pragma options inline [=] on | all | off | none |`
`0 | 1 | 2 | 3 | 4 | 5 | reset`

The `'='` is optional and no macro substitution is performed on the parameters. As with the command line, the casing of the parameters is ignored

Semantics

This pragma is used to either temporarily or permanently override the level of inlining specified by the command line `-inline` option. The syntax for the pragma following the `inline` keyword is identical to that of the command line (except for `reset` which is specific to this pragma). The parameters have the same meaning as the command line.

`on`	A function becomes a candidate for inlining when the inlining provides a speed optimization. This includes C++ functions which are not explicitly defined as inlined.
`all`	Identical to `on`.
`off`	Do not consider any functions as candidates for inlining except C++ functions explicitly declared as inline.
`none`	Identical to `off`.
`0 ... 5`	Candidates for inlining must be less than the "complexity limit" implied by the value. An inline level of 0 has the same effect as `off`. Specifying 5 means very aggressive inlining at the cost of possibly considerable code size increase.
`reset`	Reset the inline level to what was specified or implied by the command line. If no level was specified on the command line, inlining (complexity limit) is reset to 2.

As implied by the above descriptions, functions become "candidates" for inlining. There is no guarantee such functions will actually be inlined. If the compiler determines that there would be a speed benefit without excessive code expansion (guided by the specified inline option) then candidates are inlined.

When the pragma is used outside of a function, then the specified inline level overrides the command line until another `#pragma options inline` is encountered outside of any functions. If `reset` is specified, the inline level is reset to what was specified or implied by the command line.

If the pragma is placed inside a function body (i.e., anywhere between its enclosing braces), then the inline level is temporarily set according to the pragma *for that function only*. The specified inlining level applies to the entire function no matter where within the function the pragma is placed. If more than one `#pragma options inline` is placed within the function, then it's an error to specify different inlining levels. The `reset` option is not permitted when the pragma is used within functions. Following the function, the default inlining level is reset to what was in effect prior to that function.

Note, if `-share_lib_export on` or `-sym on` was specified on the command line, then warning will be issued, and the inlining level will remain unchanged.

6.3 #pragma options opt - Set optimization level

Syntax **`#pragma options opt [=] off | none | local | size | speed[,<modifier>...] | reset`**

```
<modifier> ::= unroll | norep | nointer | unswitch | unswitch_notify
```

The `'='` is optional and no macro substitution is performed on the parameters. As with the command line, the casing of the parameters is ignored

Semantics

This pragma is used to either temporarily or permanently override the level of optimization specified by the command line `-opt` option. The syntax for the pragma following the `opt` keyword is identical to that of the command line (except for `reset` which is specific to this pragma). The parameters have the same meaning as the command line.

`off`	Perform no optimization.
`none`	Identical to `off`.
`local`	Perform local optimizations and global register allocation.
`size`	Perform optimizations for size rather than speed.
`speed`	Perform optimizations for highest performance.
`reset`	Reset optimization level to what was specified or implied by the command line.

The modifier is used only with the speed parameter and can have the following values:

`unroll`	Perform speed optimizations and include loop unrolling.
`norep`	Perform speed optimizations without repeating global propagation and redundant store elimination.
`nointer`	Perform speed optimizations without interprocedural optimizations.
`unswitch`	If a loop contains an invariant branch, this option causes two copies of the loop to be made. One assumes that the condition is true, and the other assumes that the condition is false. The branch is placed before the two copies, and it jumps to one or the other copy depending on whether the branch condition is true or false.
`unswitch_notify`	This is identical to `unswitch` except that the compiler prints out a message identifying the routine where unswitching takes place. This message is output once for every branch that is unswitched.

When the pragma is used outside of a function, then the specified optimization level overrides the command line until another `#pragma options opt` is encountered outside of any functions.

If `reset` is specified, the optimization level is reset to what was specified or implied by the command line.

If the pragma is placed inside a function body (i.e., anywhere between its enclosing braces), then the optimization level is temporarily set according to the pragma *for that function only*. The specified optimization level applies to the entire function no matter where within the function the pragma is placed. If more than one `#pragma options opt` is placed within the function, then it's an error to specify different optimization levels. The reset option is not permitted when the pragma is used within functions. Following the function, the default optimization level is reset to what was in effect prior to that function.

Note, `-sym on` implies no optimization. If anything other than `off` (or `none`) is specified on the pragma, a warning will be issued, and the optimization level will remain unchanged.

7.0 Miscellaneous Pragmas

The pragmas in this category are generally unrelated to one another. They are a collection of pragmas based on enhancement requests or "deficiencies" that cannot be addressed in any other way.

7.1 #pragma unused - Disable warnings about unused locals and parameters

Syntax **`#pragma unused (var_or_param [,var_or_param] ... )`**

where, var_or_param is a local variable or function parameter.

Semantics

This pragma suppresses compile-time warnings (warning 29 and 35) that are emitted when the compiler discovers that one or more local variables or parameters have not been referenced within the body of a function definition. The pragma is only allowed within the body of a function definition.

The warnings may either be suppressed by explicitly including the pragma or specifying `-w 29` and/or `-w 35` on the command line. Number 29 suppresses warnings about unused local variables while 35 suppresses warnings about unused function parameters.

Note that variables are only checked for reference independent of flow analysis. There is no validation as to whether such variables are used correctly or skipped over due to changes in flow control.

7.2 #pragma traceback - Generate traceback table for specific functions

Syntax **`#pragma traceback [list] <func_list>`**

The syntax for the `<func_list>` is identical to that described for [no]inline_func.

Semantics

(1) This pragma is only processed if the `-tb pragma` option was specified on the command line.

(2) This pragma may only appear *outside* of any function definitions.

(3) Any function on the list must not be defined or called. See #pragma [no]inline_func for the complete semantics on the functions in the list since the segment functions have identical semantics ([no]inline_funct semantics items 1 and 2).

(4) A reference to a function in C++ implies a reference to all of its overloads.

(5) Any valid function specified in the traceback pragma that has not been defined by the end of the compilation unit will be reported as "undefined" warning.

(6) Duplicate references to the same functions are reported as a warnings.

The default for the compiler is to never generate traceback tables for functions. However, the `-traceback` and `-tb` command line options give the user control of the generation if tracebacks are desired. These command line options have the following syntax:

```
-traceback
-tb <tb_opt_list>
```

where,

```
<tb_opt_list> ::= <tb_opt> [, <tb_opt>]
<tb_opt>      ::= on | pragma | export | outofline
```

When just `-traceback` is specified then a traceback table is generated for every function. A `-tb on` is equivalent to the `-traceback` option.

The `-tb pragma` enables the traceback pragma and causes only the specifically specified functions to have a traceback table.

When `-tb export` is specified, then functions marked for exporting with #pragma export will have a traceback table.

A traceback table for a function is normally placed at the end of that function's code. These tables are not necessarily small. For something like a set of C++ member functions, which may be generally fairly small, the traceback tables could make up a considerable percentage of the total amount of code space. In a paging environment this could affect performance by needlessly spreading out an otherwise compact set of related functions. The `-tb outofline` option addresses this problem.

When `-tb outofline` is specified on the command line, all the traceback tables generated as a result of the other traceback options and pragma are generated in their own private csect (XCOFF type XMC_TB). One csect is generated for each function to which the pragma or the option is applicable. In addition , an extra branch instruction is generated, after the branch and link instruciton that symbolizes the end of a particular functions c-sect. The offset to the traceback csect from this branch instruction is encoded in this branch instuction. The offset also has a relocation entry so that the offset entry is updated if the traceback table is relocated by the linker. The linker should coalese csects of type XMC_TB and put them at the end of the PEF code section.

7.3 #pragma ignore id - Ignore pragmas

Syntax **`#pragma ignore id,...`**

Semantics

This pragma indicates that any pragma in the source specified as “#pragma id ...” is to be totally ignored by the compiler. No warning will be given. Note that the id’s may also be specified by the -ignorepragma command line option.

7.4 #pragma disjoint - Specify mutually exclusive variables and pointers

Syntax `#pragma disjoint (<disjoint-list>)`

where, `<disjoint-list>` is defined as follows,

```
disjoint_list>  ::= <disjoint-name> ',' <disjoint-name> |
                    <disjoint_list> ',' <disjoint-name>

<disjoint-name> ::= <id> | <disjoint-ptr>

<disjoint-ptr>  ::= *<id> | *<disjoint-ptr>
```

Each identifier must be defined at the point this pragma is specified. For C++, a ‘::’ may be used to specify a global scope reference. As indicated in the symtax, there must be at least two identifers or pointer specified and no duplicates are allowed. There cannot be any uses of the identifiers prior to the pragma and the indirection level (number of *) of pointer identifiers cannot be greater than the level indicated at the definition.

Semantics

This pragma informs the compiler that none of the identifiers listed share the same physical storage. If any identifiers share the same physical storage, the pragma may give incorrect results. You can use `#pragma ignore` to turn off `#pragma disjoint` to analyze the problem further. The disjoint pragma is applied to the identifiers within the scope of their use.

The identifiers cannot refer to a member of a class, structure or union, a class, structure or union tag, an enumeration constant, a label, a function or a function pointer.

Example:

```
foo(int p , int q, int r)
{
   int *s,*t;

   #pragma disjoint (*s, *t)
   #pragma disjoint (*s, p, q)
   #pragma disjoint (*t, r)

   s = &r;

   r= p + q;               // redundant store to r
   if (p == q)
      t = &p;
   else
      t = &q;

   *t = p - q;
   *s = p - q;             // 2nd store to r making the 1st redundant
   bar(&s, t);
}
```

In the above example, `s` only points to `r`, and `t` may point to `p` or `q`. The first store to `r` is redundant, since `r` is stored into again through a pointer dereference of `s` and there are no uses of `r` (directly or through `s`) after the first store. The pragmas tell the optimizer that `s` and `t` do not point to same storage and `s` cannot point to `p` or `q` and `t` cannot point to `r.` This helps the global optimizer delete the first store into `r` as a redundant store.

7.5 #pragma precompile_target, #pragma dump - Specify name of precompiled header

Syntax `#pragma precompile_target "filename"`
`#pragma dump "filename"`

Semantics

The “dump” pragma is identical to the “precompile_target” pragma. The former is the style used by the “old” MPW C compiler while the latter is for Metrowerks compatibility.

The filename supplied by these pragmas will be the file that will receive the precompiled header. If the filename is supplied as a null string, or the pragma is not present in the file, the name of the source file is used with the extension removed. If more than one pragma appears in the source, the one closest to the end of the file is used. If the filename is a partial path name, the directory used is the same as the source file. A full pathname of course specifically indicates where the precompiled header is to “go”.