Release Notes

MrC and MrCpp v. 3.0.1

C and C++ Compilers for Power Macintosh

Contents

Major Changes from v. 2.0

Warning: Incompatible changes in MrCpp's ABI (Application Binary Interface) mean that most people should recompile all projects that use MrCpp. This also means that you cannot mix C++ objects produced by MrCpp v. 3.0.1 with object files compiled from earlier versions of MrCpp. This restriction specifically applies to the use of objects produced by MrCpp compilers before version 3.0.0d1, including v. 2.0.1f (which was made available on ETO #22). Note that this only affects C++.

Language Changes

- A new C++ *vtable* layout and an ABI that is incompatible with previous versions of this compiler.
- Support for C++ Exception Handling
- Support for 64-bit integers: `long long` predefined type.
- support for C++ Run Time Type Information (RTTI).
- `bool` predefined C++ data type.
- Function-try-block syntax is now supported.
- Exception-specifications (throw clauses) are now supported on function pointer declarations.

Options

- Command line options: -alloca, -ansifor, -bool on|off, -EH, -ER, -exceptions on|off, -inline all,-opt speed, warn_uninit, -opt speed, warn_maybe_uninit, -opt size,warn_uninit, -opt size, warn_maybe_uninit and -rtti on|off, -abi [old | newest, -aslm, -ir pathname[,...], -target 704.
- The built-in boolean function __option(<keyword>) is used, for a defined set of keywords, to test whether a command line option or pragma is in effect.

  The following keywords have been added:

New __option keywords	Condition tested
`__option(RTTI)`	`-rtti on`
`__option(fp_contract)`	`-fp_contract on`
`__option(maf)`	Same as `__option(fp_contract)`
`__option(direct_to_SOM)`	`-som`
`__option(SOMCallOptimization)`	`#pragma SOMCallOptimization on in effect`
`__option(SOMCheckEnvironment)`	`#pragma SOMCheckEnvironment on in effect`
`__option(ldsize128)`	`-ldsize 128`
`__option(ANSI_strict)`	`-ansi strict in effect`
`__option(bool)`	`-bool on`

New Defaults

- The access of template functions instantiations has been reversed. The default template access is now “public”. It was previously “static” and required a `#pragma template_access` public in order to get the templates instantiated

- RTTI is on by default. Therefore, you must either link with MrCPlusLib.o or explicitly turn RTTI off by using the option `-rtti off`. (NOTE: This might be needed even if your program previously did not require MrCPlusLib.o at all.) If you used the default setting for RTTI, your C++ code MUST be linked with the MrCPlusLib.o, 3.4.4b1 (or greater version number), library. The implicit option, `-rtti on`, is the default behavior of the releases of MrCpp version 3.0.0d1 or greater . Therefore, the link to the MrCPlusLib.o is needed to avoid unresolved references by the PPCLink tool to the RTTI `type_info()` functions that are in code generated for Class objects. Another alternative is to explicitly turn off RTTI by the use of the option `-rtti off`.

  The symptoms you’ll see if you either have the wrong library or omit the library are PPCLink errors like this: “Reference to unresolved symbol `__Type_info::_rttivtbl`”. NOTE: If you had a program that formerly did not require MrCPlusLib.o but did define polymorphic types, then it will now require the current MrCPlusLib.o (or the `-rtti off` workaround).

Pragmas

- New MacSOM pragmas:

  `#pragma SOMModuleName` id_1`::`id_2`::...::`id_n,
  `#pragma SOMCheckEnvironment on | off | reset`, and

```
#pragma SOMCallOptimization on | off | reset
#pragma SOMCModuleName id1::id2:.....:idn
```

- Other new pragmas:

```
#pragma ignore <id> ...
#pragma disjoint (<disjoint-list>)
```

Intrinsic Functions

- Support for `alloca()` built-in intrinsic function.
- Intrinsic functions were added for machine instructions MTFSB0 and MTFSB1:

```
extern void __mtfsb0 (unsigned int crbD); /* MTFSB0   crbD */
extern void __mtfsb1 (unsigned int crbD); /* MTFSB1   crbD */
```

Misc.

- DOS, Unix, and Macintosh formatted files are accepted.

  Source files from all three of these platforms are accepted. The compiler assumes a "line" in a file can be terminated by a newline (`0x0D`, as in MPW), linefeed (`0x0A`, as in Unix), or the sequence newline/linefeed (`0x0D0A`, as in DOS).

General Information

MrC and MrCpp are C and C++ compilers which generate PowerPC XCOFF object files for Power Macintosh systems. Elsewhere in this document we often refer to both compilers by the single name MrC since the compilers are essentially the same other than the language dialect they support.

This release of MrC includes:

- support for C++ exception handling
- support for C++ Run Time Type Information (RTTI)
- `alloca()` builtin function for dynamic stack allocation

- pragmas for changing options in a file
- pragmas for generating inline / out-of-line traceback tables
- optimization improvements
- loop unswitching
- warning for use of undefined variables
- bug fixes
- pragmas for Direct to SOM
- pragmas for code placement, segmentation, inlining, and CFM
- support for PowerPC 604, X704
- intrinsic functions to generate special PowerPC instructions.
- intrinsic functions header file Intrinsics.h
- `bool` predefined data
- `long long` predefined data type

Incompatibilities

Precompiled Header File Incompatibilities

Precompiled headers created by MrC are not compatible with precompiled headers created by MrCpp, and vice versa.

MrCPlusLib Incompatibilities

This version of MrCpp introduces an incompatible change in the layout of the virtual function tables (*vtables*) in order to support the new Run Time Type Information (RTTI) feature. In cases where it matters (libraries which contain any polymorphic classes), the incompatibility is detected at link time. See the “Changes to ABI/Virtual

Function Tables and RTTI" section for details. The MrCPlusLib library provided with the compiler is also incompatible with the versions of the library prior to version 3.4.4b1 and must be used only with this compiler.

Command Line Options

This release has several new command line options.

Note: Defaults are underlined.

-abi newest

generate code which conforms to newest Macintosh C++ ABI standard

-abi <u>old</u>

generate code which conforms to old Macintosh C++ ABI standard

-alloca

Recognize `alloca` as a built in function. The `-alloca` and `-exceptions on` options may not be used together. See the section on new language features for a further description of `alloca`.

-ansifor

Limit for-statement declared initializer to the for-body.

-aslm

For ASLM compatibility, reverts to old vtable format. This option is incompatible with exceptions and RTTI.

-bool on

Enable C++ `bool`, `true`, and `false` as standard keywords.

-bool <u>off</u>

`bool`, `true`, `false` are normal identifiers.

-EH

Same as `-exceptions on`.

-ER

Same as `-rtti on`.

-exceptions on | off

Generate the static tables needed in order to support exception handling. Also, allow `try`, `throw`, and `catch` in C++ source code.

Note: C code should also be compiled with `-exceptions on` if it is likely to call C++ code which might throw an exception.

-ignorepragma id,...

Ignore #pragmas with the specified #pragma id's.

-inline all

Functions are inlined wherever possible, regardless of cost; this was formerly `-I2`.

-ir pathname [,pathname…]

Recursively search for include files in directory specified by pathname.

-load

A change to this option removes a restriction and now a -load ph file can be used to create another -dump ph file. In other words you can have both -load and -dump on the command line.

-nomfmem

The default behavior of the compiler is that the front end uses Multifinder memory and the back end uses MPW memory. In some cases, this results in more overall memory usage due to the fact that there is no overlap in memory usage between the front end and the back end. This can also constrain other applications which require Multifinder memory.

The `-nomfmem` option enforces the use of MPW memory by the front end. A caution here is that when using precompiled headers (load dump files), the memory requirement by the front end may be exceedingly large, in which case the option is either to use a larger MPW partition or to revert back to using Multifinder memory.

-opt speed,unroll

The default for speed compile has been changed from `unroll` to `nounroll`. To enable unrolling use `speed,unroll`.

-opt speed,unswitch

This option enables the "unswitching" optimization within loops. If a loop has a conditional statement (`if B then C else D`) and the condition is loop invariant, then the optimization is as follows: Two

copies of the loop are made, one containing `C` and all statements other than the conditional statement; the other similarly contains `D`. The condition, `B`, is removed from the loops and is used to determine which copy of the loop to execute. The case of conditionals without an `else` clause is handled similarly.

-opt speed,unroll,unswitch

This is an example to show that unroll and unbswitch may be used together, that is, these options may be combined.

-opt speed,warn_uninit

-opt size,warn_uninit

The `warn_uninit` suboption of `-opt speed` or `-opt size` reports when a variable is definitely being used before it has been assigned a value. Usually, such a situation is a logic error.

Because of the effects of global optimization, the source position reported for the reference in question is not always precise. By using the variable name and approximate line number, it should be straightforward to locate the reference.

-opt speed,warn_maybe_uninit

-opt size,warn_maybe_uninit

The `warn_maybe_uninit` suboption of `-opt speed` or `-opt size` reports when a variable is possibly used before having been set. Occasionally, correct programs may be flagged by this option. For example, the use of x in the second if-statement below is flagged by `warn_maybe_uninit` even when condition1 and condition2 are equivalent:

```
int x;
if (condition1)
     x = 1;
if (condition2)
     return x;
```

Because of the effects of global optimization, the source position reported for the reference in question is not always precise. Using the variable name and approximate line number it should be straightforward to locate the reference.

-prefix pathname [,pathname…]

Include the specified file(s) prior to reading the first source file.

(a). A -prefix file can be a precompiled header file so that -load need not be used.

(b). The first #include can be on a precompiled header as long as only comments preceded it.

-rtti on | off

Enable/partially disable run-time type information. The “off” value causes a partial disabling of the Run-Time Type Information). See “Changes to ABI/Virtual Function Tables and RTTI” below for details.

-som

This option enables Direct to Som support in MrC. 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, MrCpp will generate MacSOM enabling class meta-data instead of standard C++ vtables.

-target 604

Generate code for PowerPC 604.

-target 704

This option is to tell the compiler to schedule instructions and optimize for the yet-to-be-released Exponential X704™ processor..

-tb on,pragma,export,outofline

This pragma gives the user control over the generation of traceback tables. By default the compiler does not generate traceback tables. The `-tb on` is akin to the `-traceback` option. It generates traceback tables for all functions defined in the file. The `-tb` pragma generates traceback tables for all the functions specified by `#pragma traceback` (see “pragma” in the section below). When `-tb` export is specified, then functions marked for exporting with `#pragma export` will have a traceback table. 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 he option is applicable. In addition, an extra branch instruction is generated after the branch and link instruction symbolizing the end of a particular function’s c-sect. The offset to the traceback csect from this branch instruction is encoded in this branch instruction. The offset also has a relocation entry so that the offset entry is updated if the traceback table is relocated by the linker. PPCLink should coalesce csects of type XMC_TB and put them at the end of the PEF code section.

-unique_strings

Force all string constants to be unique. The default for the compilation has been changed from generating unique strings to sharing strings.

Note: In some previous documentation for MrC the `-curdir` option was misspelled as `-currdir`. The correct form is `-curdir`.

Changes to ABI

Two changes were made in this release of MrCpp as to how the compiler passes arguments to certain functions, and how the virtual function tables are laid out. These are changes in MrCpp's ABI (Application Binary Interface) and means that most people should recompile all of their projects that use MrCpp. Note that this only affects C++ programs.

Structure Passing by Value

MrCpp now generates code using different calling conventions when there is a function which is passed an object by value. In simple cases the object is simply passed by value as happens with structures. However, if the object is an instance of a class which has a copy constructor or a destructor, then a temporary copy of the object is created and is passed by reference.

Previously the criteria for creating the temporary and passing it by reference was just the existence of a copy constructor; now this will also happen if there is a destructor. Previously if there was a destructor but no copy constructor the object would be passed by value and destructed at the end of the called routine. Now all destructors for such objects are called in the calling routine after returning from the called routine.

The only case that is actually handled differently is the case where an object passed by value has a destructor and no copy constructor. However, since code produced by previous versions of MrCpp passed the object by value and current versions of MrCpp pass the object by reference, there is the possibility of serious runtime problems if code calling such a routine is compiled with one version of MrCpp and if the routine itself is compiled with an incompatible version of MrCpp. Thus is it recommended that people recompile everything when they switch to this new version of MrCpp.

There were good reasons for making this ABI change. First, the language standard requires that the destructor for such an object be called after the full expression in which it appears has been evaluated, so our previous handling of such cases in which we called the destructor inside of the called routine was incorrect. Secondly, our previous strategy created problems for our implementation of exceptions. And lastly, this is a small step toward a standard for a common ABI for all Macintosh compilers.

Virtual Function Tables and RTTI

For each class which contains virtual functions, the compiler will generate a Virtual Function Table *(vtable)*. In this release of MrCpp, the format of the vtable has changed in an incompatible way. When an attempt to link together incompatible object files is made, the incompatibility is detected at link time, because the new vtables will have a different kind of name. A linker error message which mentions an undefined symbol with a name that starts with `_vtbl` or `_rttivtbl` indicates an attempt to link together incompatible object files.

The `-rtti off` option allows a partial disabling of the RTTI (Run Time Type Information). The virtual function table entry which normally (`-rtti on`) contains a pointer to the low-level RTTI string is replaced with a `-1`. This does not make the vtable compatible with older versions of MrC, but it can save some data space.

Vtables generated by former versions of MrCpp did not contain space for this pointer.

New Language Features

MrC 3.0.1 supports a number of new language-level features, including new pragmas and intrinsic functions.

C++ Exception Handling

When the `-exceptions` flag is given, exception handling is enabled. This permits the use of `try`, `throw`, and `catch` blocks in C++ source, and it generates some small static tables for either C or C++ code. In order for exceptions-enabled code to work, it must be linked with exceptions-enabled libraries, and must also be able to find the new shared library, `MrCExceptionsLib`. This release supports only standard C++ exceptions—an exception can be thrown only synchronously, and only from C++ code, via a `throw` statement.

The C++ language support library (`MrCPlusLib.o`) contains memory allocation routines which will work with both `-exceptions` and non-exception-aware code. (The default versions of the `new` operator, and the array `new` operator are both changed, but the calling sequences also changed in such a way that both versions can coexist as overloaded functions in one library.)

The new shared library `MrCExceptionsLib` should be placed somewhere where it can be found by any exceptions-enabled application. The standard location is in the Extensions folder within the System Folder.

In order for exceptions to be handled correctly, exceptions-enabled libraries must be used. This is especially important for “call-back routines”, called from libraries.

The default behavior of the `new` operator has changed in this release. When the `new` operator fails because no memory is available, it will throw an exception. In previous releases, it would return a NULL (zero) pointer.

Limitations

As stated above, code compiled with `-exceptions` must be linked with the new versions of the libraries `MrCPlusLib`, `MrCIOStreams.o` (if it is used), and `MrCExceptionsLib`. In addition, PCCLink 1.5 (or a higher version number) must be used to link exceptions-aware code.

The library currently makes little use of exceptions— the various forms of the `new` operator are the only library routines which will currently throw any exceptions.

All instances of `#pragma options(exceptions)` in a source file will be ignored. Exceptions can only be turned on from the command line.

RTTI (Run Time Type Information)

MrCpp's implementation of RTTI adds five new keywords: `typeid`, `const_cast`, `dynamic_cast`, `reinterpret_cast`, and `static_cast`. In order to use `typeid`, the header `<typeinfo.h>` must be `#included`.

bool Predefined Data Type

MrCpp now supports `bool` as a predefined data type. When `-bool on` is specified on the command line, the identifiers `bool`, `true`, and `false` all become reserved keywords

of the C++ language. See latest C++ Standard documentation for the semantics on how to use the `bool` data type.

Note, with this feature enabled, older versions of types.h will cause errors since `true` and `false` are defined there as `enum` values. You must use the latest version of types.h.

In order to test for the presence of this feature you can use `__option(bool)` or `__option(nobool)`.

long long Predefined Data Type

Support has been added for 64-bit arithmetic. You indicate this by declaring a variable as type `long long`. When using this data type you do not need to use the Math64 header or library. Where possible, the compiler generates all 64-bit operations directly. Anything that cannot be done directly is done by routines provided as part of PPCRuntime.o.

The `long long` data type is accepted in most every place a `long` data type would be accepted except for the following:

- When `-ansi strict` is specified on the command line.
- As a switch statement control expression or case label.
- As a constant in a preprocessor statement.
- Bit-field widths.
- As an enum value.
- As an array dimension.
- As a template value argument.

Support for the long long data type has been added to the C library. See the MPW Libraries & Interfaces Release Notes for details.

alloca() Stack Allocation

`alloca` provides a way to dynamically allocate stack space. When the `-alloca` option is specified on the command line, the compiler recognizes `alloca()` as a built-in function having the prototype:

```
void * alloca (size_t num_bytes);
```

Calling `alloca()` dynamically allocates the requested amount of memory from the stack and returns a pointer to this doubleword-aligned area. The area is freed upon return from the function, or by any other mechanism that resets the stack pointer to a parent's call frame.

`alloca()` has the advantage over `malloc()` of being much more efficient, but the user is cautioned that `alloca()` performs no checking of stack bounds and the return value of `alloca()` does not indicate whether stack space has been exhausted. The user must ensure an adequate stack region for the program (by appropriately setting the application size resource, or for MPW tools by specifying a stack size with the "setshellsize" MPW command).

A function that calls `alloca()` is not eligible for inlining since removal of the function call boundary by inlining might greatly increase the program's runtime stack requirement.

Direct to SOM

General Description

MrCpp supports 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 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 ships 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 ships 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 handles 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 virtual.

(d) All data members must be private.

(e) Non-inlined member functions must be unique independent of casing and signature. 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 protected 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 new and delete 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 supports 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 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 (method1, method2, ..., methodn)
```

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 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 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.

`#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, by default, assumes 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` 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 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)!

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%.

Pragmas (General)

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

where var_or_param is a local variable or function parameter.

This pragma suppresses compile-time warnings (warning 29) 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 on the command line. 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.

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

  <func_list>  ::=   ['('] <the_list> [')']
  <the_list>   ::=   <func> | <the_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>
```

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++.

For C++ we can have 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`).

<the_list> may optionally be enclosed in parentheses.

The keyword `list` is ignored unless it is the only identifier specified. Note that if the first member of `<the_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. It is an error to omit all function names or to have a null function name (i.e., two commas).

The Semantics of the pragma are:

- Any function specified on the function list must not be defined or called. Declarations, however, are permitted. 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.

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

- 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).

- 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.

- 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.

- 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.

- Turning inlining or optimization off from the command line overrides these pragmas.

```
#pragma [no]inline_site ['list'] <func_list>
```

The Syntax for the `[no]inline_site` pragma is identical to `inline_func` described above. Thus the keyword `list` is ignored unless it is the only identifier specified.

The Semantics of the pragma are:

- 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.

- 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.

- Inner blocks with their own [no]inline_site pragmas override enclosing block [no]inline_site pragmas.

- 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.

- 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.

- A reference to a function in C++ implies a reference to all of its overloads.

- 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.

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

```
#pragma seldom
```

The `seldom` pragma controls movement of the blocks of code containing the pragma. `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).

The Semantics of the pragma are

- The pragma may only be used within a function (brace-enclosed block) and applies only to that block and all blocks nested within it (i.e., this has the same scoping rule as `#pragma [no]inline_site`).
- The seldom 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 the rule above applies.
- The pragma applies to the entire block that contains it even if the pragma does not appear at the beginning of the block.
- Speed optimizations like inlining may be turned off in blocks marked seldom unless the optimization leads to smaller code size.

```
#pragma segment <seg_name> [ [ list ] <funclist> ]

  <seg_name>    ::= <id> | string
  <funclist>    ::= 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`.

The semantics of the pragma are:

- This pragma may only appear outside of any function definitions.
- For the list form, 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).

- A reference to a function in C++ implies a reference to all of its overloads.
- 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.
- Duplicate references to the same functions are reported as a warnings.
- 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.
- The list form defines the segments for specific functions and takes precedence over the non-list form.
- 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.
- All functions defined for a particular segment are collected together at link time to be placed adjacent to each other in the final executable.

```
#pragma  <cfm_pragma> <option>

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

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 `<name_list>` is the same as a `<func_list>` with respect to the treatment of the keyword ‘list’ and parentheses. The use of this pragma requires that you use PPCLink v. 1.5d3 or later.

Common CFM Pragma Semantics are:

- This pragma may only appear outside of any function definitions.
- Template references require that the template definition exist prior to the pragma.
- 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.
- 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).
- A reference to a member function in C++ implies a reference to all of its overloads. References to C++ class variables is not allowed.

- 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.

- 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.

- 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`.

- Subject to the specific semantics of the pragmas, “inner” nested `on` pragmas override or merge with “outer” nested `on` pragmas.

- 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`.

The Semantics of the Export Pragma are:

- 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.

- The export and import pragmas are orthogonal and may be applied independently.

- The export and internal pragmas are orthogonal for variables and may be applied independently.

- 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).

The Semantics of the Import Pragma are:

- 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.

- The import and export pragmas are orthogonal and may be applied independently.

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

The Semantics of the Internal Pragma are:

- 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.

- Internal does affect file scoped (static) functions indicating that their function descriptor (“TVector”) may be omitted.

- It is an error to take the address of a “internal” function.

- The internal and export pragmas are orthogonal for variables and may be applied independently.

- 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).

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

The following explains the usage of these pragmas

- Terminology

    - A compilation unit is a file compiled using MrC. A fragment is formed by linking together compilation units. Global symbols in the fragment have the following attributes:

        - Export: List of global symbols that are defined in this fragment and thus can be exported

        - Import: List of global symbols that are referenced indirectly in the fragment. These symbols can either be defined in the fragment or can be defined in some other fragment. An extension of this concept is patchability. Symbols that are defined in a fragment but are patchable require that they be treated as imports. Thus for a symbol to be patchable, it should be declared as an import.

        - Internal: List of global symbols whose uses are internal to a fragment, i.e., all references to it from a fragment are direct.

- Effect of these attributes on Code generation

    - MrC default (i.e., in the absence of any attributes) is to do interprocedural analysis for functions defined in a compilation unit. Consistent with this, MrC will generate single instruction call sites (no TOC reload) for global functions defined in the compilation unit. This is also true for static functions as well

    - For functions marked internal, a one instruction call site (no TOC reload) will be generated

    - For functions marked internal and whose address is not taken, a T-Vector may not be generated in the compilation unit.

    - For data marked internal, the data may be placed in the TOC. The size of data can be controlled by a compile time option. MrC does not do this optimization yet.
    - For imported functions, no interprocedural optimizations will be done.
    - For imported functions, two instruction call sites will be generated (bl and TOC reload).
    - For imported data, access will be through TOC (that is, indirect).
- Relationship with other options
    - `shared_lib_export`: This mechanism has been used in the past to generate a list of functions in a .x file and mark them as exports. This mechanism has been extended to also mark the defined functions as imports. This will provide backward compatibility and ease of migration for users who want every function defined to be patchable. However, users should be aware that marking functions as imports turns interprocedural optimization off for such functions. For this reason, the users should migrate to using the pragma where they can selectively specify imports.

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

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. The `reset` option resets the inline level to what was specified or implied by the command line.

The semantics of the pragma are:

- 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.

```
#pragma options opt [=] off | none | local | size |
                    speed[, <modifier> ...] | reset
<modifier>    ::=   unroll | norep | nointer | unswitch
```

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 on the command line.

The semantics of the pragma are

- 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.

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

The semantics of the pragma are:

This pragma generates traceback tables for specified functions on the list. The syntax for the `<func_list>` is identical to that described for `[no]inline_func`.

The semantics of the pragma are:

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

- This pragma may only appear outside of any function definitions.

- 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).

- A reference to a function in C++ implies a reference to all of its overloads.
- Any valid function specified in the traceback pragma that has not been defined by the end of the compilation unit will be reported as an “undefined” warning.
- Duplicate references to the same functions are reported as a warnings.

`#pragma ignore <id>...`

This pragma specifies that any subsequent pragma whose id is on the given id list is to be ignored by the compiler without warning. Note that the same effect is produced by use of the `-ignore pragma` command line option.

`#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 syntax, there must be at least two identifiers or pointers specified and no duplicates are allowed.

The semantics of the pragma are:

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.

The following transformation kernel was run for 1000 iterations with 500 transforms per iteration with and without the use of the pragma disjoint . Without the use of the pragma, the program runs in 110 milliseconds giving a cycle count per transform of 29. With the use of the pragma, the program runs in 59 milliseconds or 16 cycles per transform. The hardware used is a 132 MHz PPC 604.

```
void TransformVectors1 (float *pDestVectors, float
const(*pMatrix)[3],float const *pSourceVectors, int
NumberOfVectors)
```

```
#pragma disjoint(*pDestVectors, *pSourceVectors, *pMatrix,
**pMatrix)
{
int Counter, i, j;
  for (Counter = 0; Counter < NumberOfVectors; Counter++)
  {
        for (i = 0; i < 3; i++)
        {
              float Value = 0;

              for (j = 0; j < 3; j++)
              {
                    Value += pMatrix[i][j] * pSourceVectors[j];
              }
              *pDestVectors++ = Value;
        }
        pSourceVectors += 3;
  }
}
```

Note: The MacSOM pragmas are to be found in the Direct to SOM section above.

Intrinsic functions

Support for intrinsic (built-in) functions which generate special PowerPC instructions has been added. Calling an intrinsic evaluates the function arguments into the source operands of the instruction and returns the destination operand of the instruction (if any). The use of these intrinsics does not incur any function overhead, so they offer benefits to performance-critical code that cannot be achieved with library routines.

With exceptions noted below, the function's name and prototype follow from the machine instruction's name and operands. The function return value corresponds to the instruction's destination operand, e.g. FABS; the function returns void if there is no destination operand, e.g. STHBRX. The function’s arguments agree in number and correspond with the instruction's source operands. The correspondence is also shown by the function argument names in relation to the instruction template in the comment next to each function.

The interface for these functions is as follows:

```
int __cntlzw (unsigned int rS);                    /*  CNTLZW  rA,rS */
void __dcbf (void * rA, int rB);                   /*  DCBF  rA,rB */
void __dcbt (void *rA, int rB);                    /*  DCBT  rA,rB */
void __dcbst (void *rA, int rB);                   /*  DCBST  rA,rB */
```

```
void __dcbtst (void *rA, int rB);                          /*  DCBTST  rA,rB */
void __dcbz (void *rA, int rB);                            /*  DCBZ  rA,  rB */
void __eieio (void);                                       /*  EIEIO   */
double __fabs (double frB);                                /*  FABS  frD,frB */
double __fmadd (double frA,double frC,double frB);         /*  FMADD  frD,frA,frC,frB */
double __fmsub (double frA,double frC,double frB);         /*  FMSUB  frD,frA,frC,frB */
double __fnabs (double);                                   /*  FNABS  frD,frB */
double __fnmadd(double frA,double frC,double frB);         /*  FNMADD  frD,frA,frC,frB */
double __fnmsub(double frA,double frC,double frB);         /*  FNMSUB  frD,frA,frC,frB */
float __fmadds(float frA,float frC,float frB);             /*  FMADDS  frD,frA,frC,frB */
float __fmsubs(float frA,float frC,float frB);             /*  FMSUBS  frD,frA,frC,frB */
float __fnmadds(float frA,float frC,float frB);            /*  FNMADDS  frD,frA,frC,frB */
float __fnmsubs(float frA,float frC,float frB);            /*  FNMSUBS  frD,frA,frC,frB */
double __frsqrte (double frB);                             /*  FRSQRTE  frD,frB */
float __fres (float frB);                                  /*  FRES  frD,frB */
double __fsel (double frA,double frC,double frB);          /*  FSEL  frD,frA,frC,frB */
double __fsqrt (double frB);                               /*  FSQRT  frD,frB */
float __fsqrts (float frB);                                /*  FSQRTS   frD,frB */
unsigned int __lhbrx (void *rA, int rB);                   /*  LHBRX  rD,rA,rB */
unsigned int __lwbrx (void *rA, int rB);                   /*  LWBRX  rD,rA,rB */
double __mffs (void);                                      /*  MFFS  frD */
void __mtfsb0 (unsigned int crbD);                         /*  MTFSB0 crbD */
void __mtfsb1 (unsigned int crbD);                         /*  MTFSB1 crbD */
int __mulhw (int rA, int rB);                              /*  MULHW  rD,rA,rB */
unsigned int __mulhwu (unsigned int rA, unsigned int rB); /*  MULHWU  rD,rA,rB */
double __setflm (double frB);                              /*  MFFS  frD; MTFSF 255,frB */
void __sthbrx (unsigned short rS, void *rA, int rB);       /*  STHBRX  rS,rA,rB */
void __stwbrx (unsigned int rS, void *rA, int rB);         /*  STWBRX  rS,rA,rB */
void __sync (void);                                        /*  SYNC  */
```

Other vendors provide the same intrinsics under different names. You should map names to the ones recognized by MrC.

Motorola mcc

```
#define __builtin_eieio __eieio
#define __builtin_isync __isync
#define __builtin_sync __sync
```

IBM XLC

```
#define __iospace_eieio __eieio
```

Notes:

- `__frsqrte, __fres, __fsel` are 603 and 604 instructions only.
- `__fsqrt and __fsqrts` are 604 instructions only.
- `__setflm` expands to two instructions: mffs to retrieve the old FPSCR, and mtfsf to set the PFSCR to a new value. The old FPSCR is returned. Only the bottom 32 bits of the argument and return value are defined.

- The data cache instructions and load/store byte reversed instructions take two register operands which are added to form the effective address. The operands are symmetric; however the intrinsic prototypes for the corresponding functions are given as `(void *, int)` to make sense in terms of C address arithmetic.

- The data cache instructions act on cache blocks, whose size is processor-model dependent. The dependency is especially manifest in `__dcbz` which modifies data.

- Fused multiply-add and related numerical functions are provided so that numerical code which is sensitive to the increased accuracy of these instructions can explicitly control their use. These functions would typically be used in conjunction with a compiler switch that suppresses automatic MAF generation. In the same spirit of providing explicit control, the negative MAF forms and `__fnabs` are included.

- Intrinsics.h

  Introduced at the time of the ETO22 Pre-release was a MrC-specific header file "Intrinsics.h", located in the Cincludes folder. This file is primarily for the documentation of intrinsic functions recognized by MrC version 2.0 and later. It is not necessary to include Intrinsics.h if the file is compiled with MrC, because the prototypes are built into the compiler. If the file is compiled using a compiler other than MrC, or prior to MrC version 2.0, one of two situations is possible. The other compiler may support the intrinsic as does MrC and inline it appropriately. Or, you can include Intrinsics.h for the compilation and link with the static library PPCCRuntime.o; the intrinsic function call will then resolve to an external library routine which implements the functionality.

Optimization

Loop optimizations have been improved in MrC 3.0 with the addition of unswitching on loop-invariant IF conditions (`-opt speed,unswitch`). Loop unrolling can now unroll loops which consist of multiple basic blocks. The default for -opt speed is not to unroll loops in general but unrolling is done by default for simple cases where the iteration count is a small constant. In practice this often yields the performance benefit without the size cost of unrolling all loops.

Conditional expressions of the form (`(i relop 0) ? x : y`) without side effects, and the equivalent sequence written using IF statements, are optimized as expressions and generated as a branchless inline sequence.

Pattern recognition of shift, mask, rotate and bit insert operations is improved, resulting in better use of the `RLWINM` and `RLWIMI` instructions.

Local register allocation has been improved, resulting in fewer spills.

Multiplication of integers by constants is reduced to shift and add operations when the resulting sequence improves performance. The settting of `-target` is considered in estimating the tradeoff. At -opt size, only multiplication by a power of 2 is rewritten as a shift.

Better code is generated for expressions involving char and short types. The optimizer removes unnecessary conversions to full-width integers. Optimization of long long types has been improved since the ETO #22 Pre-release.

`#pragma disjoint` allows the user to specify pointer aliasing information that cannot be expressed otherwise in C. See the description of this pragma for an example of its use in performance tuning.

A stack frame is no longer created for leaf functions with stack usage of 220 bytes or less (“red zone” optimization). At `-opt size`, functions which use more than a few non-volatile floating point registers call subroutines to do the save and restore.

The optimization improvements in MrC 3.0 are reflected in a variety of standard C and C++ CPU benchmark programs. On the cross-platform BYTE benchmark MrC has improved from an integer score of 2.8 to 6.1 on a Power Mac 9500/132. MrC’s Nullstone score of 85% is among the best of current PowerPC compilers, as is its score of 1.0 on the Stepanov C++ benchmark. MrC performance measurements are available via the web at http://www.devtools.apple.com/compilers.

Other Improvements

Error Reporting

Error reporting has been “fixed” to format type conversion errors more properly (a separator line was in the wrong place) and to show which argument is in error when a conversion error occurs in parameter passing.

New Warnings

- warning 41: This usage is deprecated. [Arises from the usage `b++`, where `b` is a boolean.]
- warning 40: Floating point constant does not fit in type xxxx. [Attempting to initialize a variable of type xxxx with a value which it cannot hold.]
- warning 39: Floating point constant represented as denormalized. [Specified floating point constant can only be represented as denormalized.
- warning 38: You can't ignore #pragma ignore! [Arises from using the option `-ignorepragma ignore.`]

Compatibility and Usage Issues

- Under certain circumstances MrC does not generate a v-table for a C++ class when it is necessary. This problem occurs if the first member function of a class is not defined. The problem may be avoided by ensuring that the first constructor for a class is defined.

  In some sense this behavior is not a problem. Instead it is a reflection of the mechanism used by many C++ compilers to decide in which compilation unit to emit the definition of a class's v-table. The convention is to do this in the compilation that includes the definition of the first member function of the class.

- Since the default for string constants has been changed from generating them uniquely to sharing them, `-unique_string` option should be used if such constants are being modified in code.

- We recommend that users of `-shared_lib_export` move to the `#pragma export` form for identifying exported symbols. This information is then internally passed to the linker in the object file and there is no need to identify exported symbols through the `-export` or `-@export` PPCLink option.

- When multiple conflicting options are present on the command line, the option that textually appears last wins. For example

  ```
  MrC -opt speed -opt local -inline off foo.c
  ```

  In this case the file will be compiled at `-opt local` without any inlining.

- The commando interface to the compiler is not complete with respect to new options added to the compiler. However, all the options supported in the commando are correctly implemented.

- Users of the Standard Template Library (STL) should be aware that MrCpp is not compatible with all of the public domain libraries or with the Metrowerks STL library.

- A common problem that has arisen in the past is to compile a function and then to link it as an export using the PPCLink option -export or @export. A side effect of this can be that the compiler will optimize a call site to such a function to one word call site, that is, one without the TOC reload slot, and PPCLink will add glue code for this call site. This can lead to erroneous run time behavior. The correct thing to do in this case is to use #pragma import for functions that need patching. The compiler will then generate a TOC reload after the call. An alternate but less optimal way to achieve the same effect is to use the `-shared_lib_export` option with the compiler. This would make all global functions defined in the file as imports. This would also turn off interprocedural analysis on those functions.

Known Outstanding Bugs

- An expression involving the long long type will generate incorrect code in the following case: a comparison result is converted to long long and used in a binary expression, e.g., the expression has the form x + (long long)(a != b). The conversion to long long is incorrectly optimized away.

Bug Fixes in v. 3.0.1

- MrC crashes when compiling MD5 stack overflow in CG caused by LO creating a huge tree. Stack check macros were added at strategic places in the backend to avert a crash when compiling programs that produce deep expression trees. (Radar #115688)

- MrC was generating the wrong code when optimization = on. The compiler overwrote a CR-reg before using it. (Radar #115691)

- MrC Preprocessor has an error preprocessing the macro expansion when it sees the ‘##’ operator. The behavior of these compilers is different than the ANSI C spec. (Radar #115738)

- MrC reports error, “cannot open PIL file 'Moonlighting:System Folder 7.5.1:Preferences:MPW:TempTS:MPEGVideoAdvanceToStartCodes.cp.n': Invalid argument.” It was not detecting when the file name was too long. (Radar #1603713)

- BE: Incorrect code motion of __setflm intrinsic by GOptimizer and scheduler MrC incorrectly moves the __setflm intrinsic backwards within the basic block for sqrt.c. This has been fixed; changed the tree builder to put setflm calls into blocks by themselves. All code motions stopped across these kinds of intrinsic calls. (Radar #164070)

- Linker error for unresolved __Type_info::_rttivtbl. This has been fixed; we have clarified the release notes for this release (starting with the 3.0.0d3 engineering release), emphasizing the need to use the MrCPlusLib.o that comes with the -rtti compiler. (Radar #1605054)

- Code Warrior Plugin compilers: MrC plugin currently drops temporary files in the source repository. This has been changed in the 3.0.1 release of the MrC plugin. (Radar #1605458)

- Code Warrior MrC/MrCpp Plugins: Does not handle UNIX-style source files (EOL == newline). The change was to “enhance” the compilers to accept Mac (newlines, 0x0D), Unix (linefeeds, 0x0A), and DOS (newlines-linefeed in that order) all as end of line characters. Both the MPW and future plugins will have this capability. (Radar #1605466)

- MrC needs intrinsics to set single bits of FPSCR. I’d like to see intrinsics to support the mtfsb1 and mtfsb0 commands. This has been chnaged and the “new” intrinsics implemented. (Radar #1605458)

  ```
  void __mtfsb0(unsigned int);
  void __mtfsb1(unsigned int);
  ```

- MrCpp cannot invoke a virtual function of a base class defined within another class. (Radar #1606815)

- MrC has a long double optimization bug. Problem ws fixed by changing GRA. In some cases the second part of a long double register candidate was participating in data flow analysis. (Radar #1606904)

- Error message information lost. Multi-line error messages are no longer truncated. (Radar #1610766)

- Two pointer updates do not generate "lbzu" instructions in tight loop. Bugs prevented this case from being recognized. (Radar #1611376)

- FE: symbol scoping problem with function-try-blocks; only the parameters are visible in the catch blocks. (Radar #1611546)

- MrC doesn't honor #pragma import with -opt speed. (Radar #1612002)

- Extra warning emitted with `-shared_lib_export on` and `-sym on`. This was because the option, `-shared_lib_export on` implies `-inline 0` and was “pretendening” that an EXPLICIT `-inline 0` was done. You would/should get the

warning when the “-inline” is explicitly stated (other than 0). I suppressed the "explicit" switch in “-shared_lib_export” on so the warning will not appear when `-shared_lib_export on` is done. (Radar #1613418)

- Unsigned cast lost in conversion of 32-bit integer to long long. (Radar #1616590)
- "function-try-block" not yet implemented. Support for this has now been implemented. (Radar #1363228)
- FE: Compiler Error building ODF. This problem occurs with exceptions and sym on when there is a precompiled header which includes classes with inline member functions which use operator new. (Radar #1600614)
- An int and a throw in a conditional (?:) expr has a problem. The fix also makes sure that both the second and third operands of the ternary are not throws. (Radar #1363226)
- MrCpp rejects exception spec in func ptr declaration. (Radar #1364821)
- Compiler fails to handle redefinition of a template function. (Radar #1401810)
- SCpp and MrCpp generate bad code for 'char(x)' type of casting in switch. The compiles can disambiguate the cast vs. declaration ambiguity in this test. (Radar #113087)
- MrC treats func ptr param decl with same name as class treated as an expr.This has been fixed. The routine that tries to decide whether it has a declaration/cast or expression now is more syntax sensitive to avoid such mistakes. (Radar #1191289)
- enum & int overloading and argument matching error. Compiler now does the promotion to int of any enum involved in an expressions. (Radar #1191051)
- A union may not have virtual functions. This has been fixed and the compilers report an error. (Radar #1215450)
- MrC nested scope bug. Externs are declared in the block in which they occur. A global copy is also made and that is the one actually used for code generation. (Radar #1174649)
- Enhancement request: exploit “red zone” for locals and temporaries. Changed to use the red zone of the stack for leaf functions whose frame size is 220 bytes or less. (Radar #1339572)
- do / while loop does not use BDNZ and CTR. (Radar #1393519)
- Overloading does not consider best match in local scope first. (Radar #1172083)

- Accepts taking an address of cast expression that does not yield a lvalue. (Radar #1224842)
- Two classes in different scope with same v-base name get duplicate vtable entry. (Radar #1197954)
- Pascal keword in overloaded operator function gets error. (Radar #1191286)
- "c %= x" is different from "c = c % x"; first is unsigned, latter is signed. (Radar #1394695)
- The compiler generates bad code, while accessing parameters of a function, if the parameters contain a long double and the total byte length of all parameters is greater than 32 bytes.
- Some problems have been detected with extracting bit fields whose size is smaller than a character. (Radar #1364763)
- Incorrect values are sometimes assigned during structure assignments of four bytes or less. (Radar #1364067)
- The compiler does not correctly handle static initialization of structures containing bit field groups which are smaller than a word. A bit field group is a group of successive bit field declarations that are packed together. One manifestation of this problem is an error message from the compiler indicating that it ran out of memory while requesting a very large amount of memory. This problem occurs with structures defined with the mac68k alignment mode. (Radars #176381, #1339273)
- In addition, bugs with the following Radar numbers were fixed: 1616141, 1616590, 1618383, 1618542, 1397516, 1398002, 1253046, 1377140, 1172086, 1360164, 1360624, 1600870, 1601252, and 1601609.