Release Notes

MrC and MrCpp v. 5.0.0d3 — Pre-release

Contents

Important: Please read the Release Notes for MrC and MrCpp v. 3.0.1 (Golden Master) for a full description of changes since ETO 22. These release notes provide information about changes since that version.

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.

The 5.x series of releases of MrC adds:

- greatly improved support for templates.
- more complete support for C++ namespaces.

The 4.x series of releases of MrC adds:

- support for C++ namespaces.
- support for the AltiVec™ architecture. AltiVec is supported in accordance with the Motorola AltiVec Programming Model document. MrC[pp]'s implementation of the Model is covered in a separate document ("AltiVec Support in MrC[pp]").

™ AltiVec is a registered trademark of Motorola, Inc.

Acknowlegements

It is one thing to put exceptional effort into a product when it is your job; it is something else to do so when it is something you do in your spare time. We would like to recognize the following honorary members of the MrC compiler team:

- Tsutomu Yoshida, who provided invaluable testing and advice during the development of MrCpp's new template mechanism
- Alex Rosenberg, who fixed a number of optimization bugs and provided a new optimization

New in v. 5.0.0d2

- MrC now implements a "noreturn" optimization. See the '"Noreturn" Optimization' section below for more details.

- The compiler now accepts arbitrary escaped characters (e.g., "`\%`"). When `-ansi on | strict` is specified a warning is issued for characters which may not be escaped according to the ANSI standard. Previously the compiler reported illegal escapes as errors. This change allows for compatibility with the original K&R C specification.

New in v. 5.0.0d1

- MrCpp's template mechanism has been totally revamped. All of the template features described in the C++ standard should be supported, with the exception of partial specialization.

- Support for the using declaration has been added, allowing individual declarations from namespaces to be used in the current scope.

- When there are definitions of inline functions in headers, the functions may not always be inlined, so code may be generated for the functions even though it is declared to be inline. By default, MrCpp creates local static copies of such functions. New "`global`" and "`static`" modifiers are now accepted by the "`-inline`" command line option, allowing inline functions to be generated as either globals or statics. The default is still static.

  ```
    -inline <level>[,global]
  ```
  or
  ```
    -inline <level>[,static]
  ```

  Note that the linker will issue warnings for duplicate non-template instance definitions when such inline functions are generated as globals. The benefit of global functions is that there will only be a single copy of each function.

  The #pragma for inlines has also been enhanced to support this new option.

  ```
    #pragma options inline <level>[,global]
  or
    #pragma options inline <level>[,static]
  ```

- MrC now generates more optimal prologs for functions whose only use of AltiVec registers is to return a value in vector register v2.

- MrC has new modifiers for the `-includes` option (and equivalent pragmas), allowing more flexibility in the handling of Unix and DOS pathnames.

  ```
  -includes unix_mac
  -includes dos_mac

  #pragma options ([!]unix_mac_includes)
  #pragma options ([!]dos_mac_includes)
  ```

  The new options are similar to the original options (`unix` and `dos`) but will treat the pathname strings as normal Mac file names if they cannot be found using the pathname as a DOS or Unix pathname.

New in v. 4.1.0a8

- MrCpp used to generate less efficient code for class initializers of the form “`b(a)`” compared to the code generated for “`b = a`”. Now the two forms are equivalent as long as there is not a user-defined copy constructor.

New in v. 4.1.0a7

- The standard function epilog code has been improved by one cycle when the function is less than 32760 bytes long and there are no condition registers to restore.

- The compile-time performance of the compiler’s handling of large numbers of initializers for an array declared with an unknown dimension (e.g., char x[] = {...};) has been improved.

New in v. 4.1.0a6

- The AltiVec model has been extended to support three new AltiVec data types: `vector unsigned int`, `vector signed int`, and `vector bool int.` These three types are intended to replace the long counterparts (`vector unsigned long`, `vector signed long`, and `vector bool long`). Use of the “old” types is now deprecated as is documented in the “Motorola Programmer Interface Manual” (which can be accessed from http://www.mot.com/SPS/PowerPC/AltiVec/facts.html). The latest version of the "AltiVec Support in MrC[pp]" document also documents these new types.

Note that while the older types are deprecated, they are still supported. No warning is given if they are used. Furthermore, the int types are treated as synonymous with the long types.

New in v. 4.1.0a5

- There have been some changes to the mangling of template names, making the output of the new compiler incompatible with that of older compilers. The unmangling library has been updated but it may take some time before support for the new mangling appears in your debugger of choice.
- The reporting of template instantiation errors has been improved and now displays the template name and arguments.
- The precompiled header format has been changed, requiring that precompiled headers be rebuilt to work with the new compiler.
- The "AltiVec Support in MrC[pp]" document has been updated to describe changes in the AltiVec programming model. Four new intrinsics have been added (vec_cmple, vec_cmplt, vec_abs, and vec_abss), and four redundant intrinsics have been removed (vec_unpack2sh, vec_unpack2sl, vec_unpack2uh, and vec_unpack2ul).

New in v. 4.1.0a3

- The mechanism which checks for uninitialized variables has been improved and can now detect more cases than before. This mechanism is invoked via the `warn_uninit` and `warn_maybe_uninit` modifiers to the `-opt speed` and `-opt size` options.
- Section 3.1 of the "AltiVec Support in MrC[pp]" document has been updated to describe a new “`@`” flag for specifying an arbitrary separator string in `printf` and `scanf`.

New in v. 4.1.0a2

- A number of new intrinsics have been added. (Radar 2287980)

```
int __rlwimi(int, int, int, int, int);
     // rotate left word immediate then mask insert
     // Note: the first argument is overwritten.
int __rlwinm(int, int, int, int);
     // rotate left word immediate then AND with mask
int __rlwnm(int, int, int, int);
     // rotate left word then AND with mask
```

- The `__lhbrx`, `__lwbrx`, `__sthbrx`, and `__stwbrx` instrinsics have been changed to allow their pointer arguments to point to a volatile object. This will produce a volatile load or store (with the appropriate restrictions on how they can be optimized).

- AltiVec loads and stores now permit a pointer to a volatile object. Such loads and stores are now typed as volatile (with the appropriate restrictions on how they can be optimized). This feature is not a part of the standard Motorola AltiVec programming model; it use will result in a warning (warning 47).

- Several new optimizations are now performed on functions using AltiVec operations. When possible in leaf functions (functions which do not call other functions) the compiler will try to save the calling function's VRsave register value in a volatile register (R3-R10). The compiler will now try to generate delayed prologs even when traceback tables are being generated.

- The "AltiVec Support in MrC[pp]" document has been updated.

New in v. 4.1.0a1

- MrC's handling of low and out of memory conditions has been improved. Such problems typically occur when compiling with `-opt speed` or `-opt size`. Since MrC compiles and generates code on a function by function basis, problems with memory are often exacerbated by the use of `-inline 5` or `-inline all` which have the effect of pulling the bodies of called functions into the function being compiled, increasing size and complexity of what must be optimized. MrC now makes better use of temporary memory available through the Finder and provides better diagnostics when out of memory conditions occur.

- MrC now recognizes more opportunities for optimizations of vector instructions.

MrC will convert some `vmr` instructions into equivalent `vsldoi` instructions to improve instruction scheduling.

For more information on MrC[pp]'s optimization of AltiVec instructions see Appendix D in "AltiVec Support in MrC[pp]".

- MrC now produces better code for `return (0 != a)`. Such comparisons are now turned into the canonical form `(a != 0)` which results in better code. (Radar 1315984)

- You may now use `__option()` in any expression instead of just preprocessor expressions.

- Long double constants are no longer aligned to 16-byte boundaries in the string table; 8-byte alignment is sufficient since long doubles are accessed as a pair of doubles.

- Two new `__option()` keywords have been added.

`altivec_model`	set if `-altivec_model on` or `-vector on` specified
`noaltivec_model`	set if `-altivec_model off` or `-vector off` specified

- A number of new intrinsics have been added.

```
float __fres(float);    // floating reciprocal estimate single
float __fabsf(float);   // absolute value of a float
float __fnabsf(float);  // negative absolute value of a float
int __abs(int);         // absolute value of an integer
long __labs(long);      // absolute value of a long integer
void __eieio(void);     // enforce in-order execution of I/O
void __sync(void);      // synchronize
void __isync(void);     // instruction synchronize
```

- A new command line option has been added to control how path names in `#include` directives are interpreted.

```
-includes dos | unix | mac
```

The default is mac.

- A new command line option has been added to control whether AltiVec recognition is enabled.

```
-altivec_model off | on[,[no][altivec_]vrsave]
```

This is an alternative to the `-vec[tor]` command line option.

- Several new pragmas have been added to control AltiVec recognition and code generation.

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

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

The `altivec_model` pragma is used to control recognition of AltiVec constructs. The `altivec_codegen` pragma tells the compiler that it is allowed to perform vectorization optimizations, converting scalar code to vector constructs where possible. (Note: this pragma currently has no effect and will be used to control future optimizations.) The `altivec_vrsave` pragma controls whether the compiler generates code to update the VRsave register. (This can also be controlled by the `[no][altivec_]vrsave` parameter on the `-altivec_model` and `-vector` command line options.) See section 2.3 in the separate "AltiVec Support in MrC[pp]" document for a more complete description of these pragmas.

- A new macro, `__ALTIVEC__`, is set to 1 to indicate that other AltiVec macros are supported.

New in v. 4.1.0d4

- A new command line option (`-longlong on | off`) has been added to control whether the compiler's built-in support for the long long data type is enabled. The default is for long long support to be enabled. Using this option will change the value of the `__option(longlong)` preprocessor function and the `_LONG_LONG` macro.

- MrC now recognizes a variety of vector instruction usages which it can replace with more optimal forms. (Radar 2258481)

  Additional vector constant optimizations have been added to increase the possibility of generating `vspltis<s>` (<s> = b, h, w) instructions instead of literals.

  The compiler now recognizes more vector constants as opportunities to be replaced by instructions that generate those constants. In cases where there is more than one way to generate a vector constant the choice is made based on what is most favorable for instruction scheduling.

  All restrictions that the optimization of vector saturate instructions have been removed. Previously these were handled less than optimally when `mfvscr` and `mtvscr` instructions were used.

  For more information on MrC[pp]'s optimization of AltiVec instructions see Appendix D in "AltiVec Support in MrC[pp]".

New in v. 4.1.0d3

- MrC now supports the `mutable` keyword.

- The format for MrC's precompiled header files has changed due to changes in the implementation of macros, so the version number with which such files are marked also has changed, requiring rebuilding of any old precompiled header files.

- Previously, when compiling under the `-opt speed` or `-opt size` option in low memory conditions, MrC would reduce the level of optimization and continue compiling. For example, if a file contained a particular large function foo, MrC might compile that function at `-opt local` while compiling the rest of the file at `-opt speed`. A diagnostic message was printed out in such circumstances. MrC has now been changed to issue an error and terminate the compilation when there is insufficient memory to perform the requested optimizations. If the old behavior is desired, a new modifier, `ok_to_reduce_opt,` may be added to the optimization option specification (as in `-opt speed,ok_to_reduce_opt`).

- Some of the AltiVec instruction scheduling problems in the previous release have been addressed.

- Two new `__option` keywords have been added.

vector	set if -vec on specified on command line
novector	set if -vec off specified on command line

New in v. 4.1.0d2

- AltiVec is supported in this release in accordance with the Motorola AltiVec Programming Model document. MrC[pp]’s implementation of the Model is covered in a separate document ("AltiVec Support in MrC[pp]").

  Note that the MrC instruction scheduler simulates a processor with a single vector unit that has single stage execution. This does not match any present or future implementation of the AltiVec architecture. In other words, at the present time, the scheduling is not optimal for the AltiVec instructions. We are looking into fixing this in a future release.

- Stricter prototype checking.

- New `__option` keywords added for Metrowerks compatibility.

mpwc_newline	same as mapcr

preprocess	set if -c specified on command line
traceback	set if -tb on or -traceback specified on command line
wchar_type	0 for C, 1 for C++

- Intrinsic functions for PowerPC operations now participate in more optimizations.

- MrCpp now performs the "empty member optimization", that is, base classes which have no data will occupy no space (even though the base classes by themselves must have a non-zero size).

New in v. 4.0.0d1

- Namespaces are supported, including unnamed namespaces, nested namespaces, namespace aliases, and using directives. Using declarations are not supported. (Using directives allow declarations in the specified namespace to be referenced without qualification in the scope with the using directive. Using declarations allow a single declaration from a namespace to be referenced in the current scope without qualification.)

- All keywords defined in proposed C++ standard are now treated as such and may not be used for identifiers. The keywords now recognized are `namespace`, `using`, `mutable`, `explicit`, and the alternative operator representations (`and`, `and_eq`, `bitand`, `bitor`, `comp`, `not`, `not_eq`, `or`, `or_eq`, `xor`, and `xor_eq`).

- Of these keywords, the only one not currently supported is `mutable`. Namespaces are described elsewhere. The alternative operators can be used wherever the standard operators can be used. The `explicit` keyword is also supported.

- The `-alias` option controls whether MrC[pp] will perform pointer alias analysis. This option is discussed below in the section "Pointer Alias Analysis Optimizations".

Using MrC's Optimization Options

MrC and MrCpp provide a variety of options to control the optimization of generated code. Producing the most optimal code for your particular program is not always as simple as setting all of the controls to their most aggressive settings.

In addition to the `-opt speed` and `-opt size` options which provide for a variety of aggressive optimizations oriented either toward fast performance or small size, there are several optimizations that can be controlled individually. The goal of the more generic

`-opt speed` and `-opt size` optimization options is to provide a generally optimal set of optimizations that apply well to a broad variety program code. The more specific optimization options provide for control of optimizations whose benefit will vary more depending on what is being compiled.

The `-opt speed` option provides a variety of modifiers for controlling individual optimizations. The syntax for the use of these modifiers is to have them follow `speed` in `-opt speed` separated by commas with no intervening spaces, as in `-opt speed,unroll,unswitch`.

The `unroll` modifier instructs the compiler to do loop unrolling where possible, which entails expanding loop code to perform several interations each time around the loop reducing the number of tests for loop termination. The end result is bigger code which executes fewer tests. This can be done when the compiler can make inferences about the number of loop interations.

The `unswitch` modifier instructs the compiler to do loop unswitching, which entails identifying tests inside of loops which will not change during the evaluation of the loop so that the loop can be transformed into a test followed by two copies of the loop, one with the code corresponding to the test being true and the other with the code corresponding to the test being true. The end result is bigger code which executes fewer tests.

The -inline optimization options controls the process of inlining, which replaces function calls by the actual code for the function, eliminating the procedure call overhead but expanding the size of the function doing the call. Various levels of inlining are supported, with higher levels inlining code for increasingly large functions. In addition to the benefit of removing procedure call overhead, inlining can open up new possibilities for further optimizations when the code for the inlined function is merged into the code for the calling function. On the down side, at higher levels of inlining functions can become so large that there may be undesirable effects on cache performance. Overly aggressive inlining may also result in less optimal register assignments within a function in which calls have been inlined due to more competition for registers. As a result, specifying too high a level of inlining can have a negative effect on program performance. For this reason, you should experiment a bit in order to find the optimal inline setting for your code.

The inline option choices are `all`, `on`, `none`, `off`, `0`, `1`, `2`, `3`, `4`, and `5`. Specifying `-inline all` will cause functions to be inlined wherever possible, regardless of the cost. Specifying `-inline on` causes functions to be inlined at level 2 as described below. Using `-inline none` or `-inline off` suppresses inlining of functions. Numeric inlining values from 0 to 5 specify levels of inlining from none to all varying on the size and complexity of the functions being considered as candidates for inlining. **It is important to note that the default level of inlining for -opt speed is 2. Use of higher levels of inlining may not be advantageous.**

One side-effect of higher levels of inlining is that the size and complexity of functions is greatly increased as function calls are expanded into the actual code for the called function. Since the inlining process is recursive the original calling function may expand significantly beyond its original size. As a result, the space required to perform other optimizations can grow considerably over what is required for the default `-opt speed` setting because many optimizations require multi-dimensional tables whose dimensions depend of the number of statements or expressions in a function. Due to the increased space required to optimize larger functions, MrC may not always be able to complete the optimization of a very large function in the memory available. The `ok_to_reduce_opt` modifier to the `-opt speed` and `-opt size` options is available to indicate that the compilation should not fail in such cases but should continue at a lower level of optimization for the function in question.

MrCPlusLib.o Compatibility Requirements

There has been a change to the prototype for the `__rtti_cast` function, which is called for any dynamic_cast. Before this bug fix, any attempt to dynamic_cast a const reference or pointer would fail. Therefore, this version of the MrCpp compilers must be linked with the library, MrCPlusLib.o, 3.5d3, which is also available on ETO #23 in the folder, "Prerelease:Libraries&Interfaces:Libraries:PPCLibraries: ". This release of MrCpp is incompatible with earlier versions of the MrCPluslib.o. However, this version (3.5d3) of the MrCPlusLib.o library is compatible with earlier versions of the MrCpp compiler.

MrCExceptionsLib Compatibility Requirements

Version 4.1d1 or newer of MrCExceptionsLib is required when your program uses exceptions and AltiVec instructions. The new version of the library is required to restore the contents of AltiVec registers in the context changes that occur when an exception is thrown.

New Language Features

MrC 4.0 supports namespaces, including unnamed namespaces, nested namespaces, namespace aliases, and using directives. Using declarations are not supported. (Using directives allow declarations in the specified namespace to be referenced without qualification in the scope with the using directive. Using declarations allow a single declaration from a namespace to be referenced in the current scope without qualification.)

MrC 4.1 supports the `mutable` keyword.

Keywords for __option and #pragma

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.

==

MrC[pp] __option and #pragma options keywords

Keyword	__option true or value	#pragma options allowed
altivec_model	-altivec_model on, -vector on	yes
ansi	-ansi relaxed | strict	yes
ansi_relaxed	-ansi relaxed	yes
ansi_strict	-ansi strict	yes
ANSI_strict	-ansi strict	yes
bool	-bool on	no
chars_unsigned	-char unsigned	yes
direct_to_SOM	-som	no
dos_includes		yes
dos_mac_includes		yes
enumsalwaysint	-enum int	yes
exceptions	-exceptions on	yes
fp_contract	-maf on, -fp_contract on	no
ldsize128	-ldsize 128	no
little_endian	0 (supported in plugin only)	no
longlong	-ansi off | relaxed	no
maf	-maf on, -fp_contract on	no
mapcr	-nomapcr	yes
mpwc_newline	-nomapcr	yes
noaltivec_model	-altivec_model off, -vector off	yes
nobool	-bool off	no
nolonglong,	-ansi strict	no
novector	-vec off	no
pack_enums	-enum min	yes
precompile	-dump (or plugin precompile)	no

preprocess	-c	no
read_header_once	-notonce not specified	yes
require_protos	-proto strict	no
require_prototypes	-proto strict	no
rtti	-rtti on	yes
RTTI	-rtti on	yes
SOMCallOptimization	#pragma SOMCallOptimization on	no
SOMCheckEnvironment	#pragma SOMCheckEnvironment on	no
stdc	1 if C++, 0 of C	no
struct_align	-align value (1, 2, 4)	yes
system_includes_from_project_tree	-inclpath normal	yes
traceback	-tb on or -traceback	no
trigraphs	-ansi relaxed | strict	no
unix_includes		yes
unix_mac_includes		yes
unsigned_char	-char unsigned	yes
vector	-vec on	no
wchar_type	1	no

Pointer Alias Analysis Optimizations

The `-alias` option controls whether MrC[pp] will perform pointer alias analysis. Such analysis allows the compiler to determine the impact of pointer usage on program values. It is called alias analysis because pointers provide aliases to values in the sense of being another way access a value without referring to the value directly (as is done with a variable reference). The use of pointers can have an adverse effect on many optimizations, for example, when encountering an assignment through a pointer, the compiler must assume that some variable's value has changed (in the worst case, all variables!). The `-alias` option allows you to tell the compiler about how you have used pointers in the file being compiled, enabling the compiler to make assumptions about how values are affected by pointer usage.

The ANSI C standard states a number of restrictions on the correct use of pointers:

- An object's stored value must be accessed only by an lvalue that has one of the following types: the declared type of the object (with or without const and volatile qualifiers), a signed or unsigned variant of the former, a struct or union type that includes one of the above among its members, or a character type. Additionally, an object declared as const may not be stored.

- An integral type added to or subtracted from a pointer yields a pointer value that when de-referenced must point to an element of the same array object as the original pointer.

- Between the previous and next sequence point an object must have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value must be accessed to determine the value to be stored.

The `-alias` option has the following forms.

`-alias off`	disable pointer alias analysis (the default)
`-alias [addr]`	enable pointer alias analysis by distinct address
`    [,ansi]`	enable pointer alias analysis by ANSI rules
`    [,type]`	enable pointer alias analysis by data type (assume pointers to different types are not aliases)

The default is for pointer alias analysis to be disabled. By specifying one or more of the `addr`, `ansi`, or `type` specifiers (separated by commas) following the `-alias` option, you can tell the compiler to make one or more assumptions about how pointers are used in the code being compiled.

The `type` specifier tells the compiler that pointers to types T1 and T2 will never refer to the same memory location if T1 and T2 differ.

The `ansi` specifier tells the compiler to use strict ANSI C aliasing rules. An object with a name a declared type T1 is not read or written by a pointer to type T2 if T1 and T2 differ (according to the ANSI rules described above). A pointer assigned the address of an array object and modified solely by increment or decrement operations will access only elements of the original array object. An object is modified only once between sequence points, so multiple store instructions between sequence points are known to be distinct. And, an object modified between sequence points will not be accessed by load instructions following a store instruction. Note that using the `type` specifier implies a more restricted use of pointers than does the `ansi` specifier because `type` implies that char pointers cannot point to other types.

The `addr` specifier tells the compiler to assume the worst -- i.e., any type of pointer can point to any type of object -- in which case the compile must do a complete analysis of pointer usage based on address information rather than type information.

While the assumptions behind each of the options may sound reasonable, you should carefully consider how pointers are used in your code before enabling the `-alias` option in order to prevent inappropriate optimizations made on inappropriate assumptions. A useful place to start is with the `-alias ansi` option; use of this option setting should be safe unless you are using pointers in a way which violates the ANSI rules described above. The `-alias type` option can be used if you are sure that all pointers only point to objects of the same type as the declared types of the pointers. Alternately, `-alias addr` does a more constrained analysis of pointers without making any assumptions based on pointer types.

The use of the `-alias` option is restricted depending on the setting of the `-opt` option. At `-opt off` use of the `-alias` option is not permitted. At `-opt local` either the `type` or the `ansi` specifier may be used (but not both). At `-opt speed` and `-opt size` either the `type` or the `ansi` specifier may be used; the `addr` specifier may also be used.

"Noreturn" Optimization

It is often the case that some code paths are known to the developer to be "impossible". The "noreturn" compiler extension provides a mechanism for developers to provide hints about "impossible" code paths through source code annotations. Numerous standard and platform-specific functions calls are defined to not return to the caller. The ANSI/ISO C standard `exit()` function is one example. The Mac OS `ExitToShell()` function is another. We refer to such functions as "noreturn" functions.

Listing 1:

```
if (a == NULL) {
  fprintf(stderr, “a was NULL!\n”);
  exit();
}
```

In listing 1, any code placed after the call to `exit()` cannot be reached. However, the compiler does not "know" that `exit()` is a noreturn function. Now the compiler can informed via a source annotation.

Method 1:

```
void exit(void);
#pragma noreturn(exit)
```

Method 2:

```
void exit(void) __attribute__((noreturn));
```

Method 1 is the syntax used in compilers from Symantec, Zortech, and Apple. Method 2 is the syntax used the GCC compiler and in future compilers from Metrowerks. MrC supports both methods. Note that #pragma directives cannot be created via macro expansion, so the latter syntax is preferable.

If a function has been properly annotated with one of the above methods, then several changes takes place. One of the most visible differences is that the compiler may no longer produce a warning about the lack of a return statement on every path out of a function. Additionally, the compiler may be able to perform additional optimizations with this information.

Listing 1 may look familiar, since it is similar to how the standard `assert()` macro expands. In fact, an improved <assert.h> may include something like listing 2 for the NDEBUG case.

Listing 2:

```
#if defined(__MRC__)%%(__MRC__>=0x500)&&__option(global_optimizer)
  #define assert(expression) ((expression)?(void)0:__noreturn())
#else
  #define assert(ignore) ((void)0)
#endif
```

The `__noreturn()` function is a new feature in MrC. It is an intrinsic noreturn function that does nothing and may be used as a source annotation to indicate that a given code path is impossible. A number of optimizations are possible given the additional information provided the `__noreturn()` function.

Bug Fixes in v. 5.0.0d3

- When spilling registers, MrC would sometimes calculate incorrect stack offsets for spilled registers. (Radar 2548444)
- The use of an undefined goto label would result in an internal error.
- MrCpp would run out of memory trying to parse namespace qualification written as `NS.xxx` instead of `NS::xxx`.
- MrCpp was not generating explicit data in the object file for static const members.
- MrCpp's type checking mechanism did not always handle the "mutable" attribute correctly.
- Some uses of the __noreturn() function would cause the compiler to crash.
- MrCpp did not always take into account the namespace of a previously defined symbol when defining the same symbol in a different namespace. (Radar 2496482)

Bug Fixes in v. 5.0.0d2

- A number of template bugs were fixed. (Many thanks to Tsutomu Yoshida for his continued diligence in tracking down and reporting MrCpp problems.)

- MrCpp would become confused about constructor generation for classes including members whose names began with two underscores.

- MrCpp 5.0.0d1 included a regression that caused it to no longer accept declarations of the form "`friend class A::B;`".

- MrCpp did not handle template instance qualifiers of non-template classes, e.g., `A::B<T>`.

- The compiler was generating an extra function descriptor for forward-declared K&R style functions, as in the following example:

```
void foo();
void foo(x)
  int x;
{}
```

- MrCpp was not checking for the global scope qualifier ("::") on struct tag names, as in the following example:

```
void foo(struct ::std::S *p);
```

- The compiler would sometimes crash when `-opt speed | size` was specified and it encountered an assignment of a struct to another struct of the same type appearing as the first member of an enclosing struct, as in the following example:

```
struct A {int w, x;};
struct B {struct A wx; int z;};
A a;
B b;
b.wx = a;
```

- MrCpp did not allow operator functions having only enum or reference to enum parameters (see the C++ Standard, section 13.5, paragraph 6).

- MrCpp did not correctly handle the setting of the "this" parameter when calling member functions referenced through using declarations and when the called function was a member of a multiply-inherited base class.

- The compiler would sometimes generate two-register address instructions (i.e., base + index) incorrectly using r0 as a base register. (Radar 2455437)

- MrCpp did not properly handle default function template parameters when these parameters were defined in terms of types specified earlier in the template parameter list.

- The compiler would sometimes generate an assertion when compiling a function-level try block.

- MrCpp did not correctly handle typedef classes used to specify static members of templates.

- The compiler did not parse expressions of the form `“size of (a)[0]”` correctly.
- MrCpp had a problem with precompiled headers involving destructors for derived classes which must call virtual destructors in a base class.

Bug Fixes in v. 5.0.0d1

- The compiler was not detecting attempts to redefined typedefs for structs and classes.
- The compiler generated incorrect code for the following floating point constructs:
  ```
  x = -0.0
  x * -0.0
  _inf() * 0.0
  f(x) * 0.0
  ```
  (Radar 2363288)
- Errors involving template instantiations and friend declarations did not always display the line on which the error occurred.
- When trying to optimize operations on structures (with the `-opt speed` or `-opt size` option in effect) the compiler could go into a loop and eventually overflow the stack.
- The compiler did not detect some invalid casts, resulting in an internal compiler error of the form:
  ```
  ### addr_trans(…): Unexpected operator (…)
  ```
- MrCpp was overly restrictive in what it accepted for return types in declarations of `operator->` functions.
- The compiler sometimes mishandled copy constructors, resulting in an internal compiler error of the form:
  ```
  ### addr_trans(…): Unexpected operator (…)
  ```
  This problem could result from a compiler-generated call to a copy constructor, e.g., when the compiler generates a temporary object when passing an object by value.
- MrC mishandled the AltiVec `vec_dssall()` function, only generating correct code for the first instance of the function appearing in the program.
- Some uses of the AltiVec `vec_dst()` and `vec_dstst()` functions could lead to asserts in the compiler.

- In particular circumstances the compiler would assert while trying to optimize AltiVec load and store operations.

- MrCpp did not support “Koenig lookup” which allows namespaces referenced in a functions parameter list to be used in the body without explicitly using the namespace.

- The compiler did not handle struct and class tag with namespace qualification.

- MrCpp did not always consider open namespaces when searching for symbols.

- The compiler mishandled passing long long parameters to functions taking variable argument lists.

- MrCpp was not always able to find constructors which were declared to be explicit.

- A problem was introduced in MrCpp 4.1.0a8 which did not allow assignments to dereferenced const pointers. (Radar 2364467)

- MrCpp did not support default type arguments for templates. It mishandled non-type parameters. It did not handle template template parameters. It did not allow nested templates. It did not handle non-inlined member functions of classes nested inside of class templates. It instantiated templates which were not actually needed. It did not recognize many cases of valid template syntax, and it asserted while trying to handle others. The template mechanism has been rewritten. (Radar 1327995, 1622458, 2338822, 2340563, 2406076)

- Given a declaration of a multi-dimensional array type, if that type was used in the declaration of a function argument with the const attribute, the array type would subsequently behave as if it had been declared with the const attribute.

  Example:
  ```
    typedef int A[4][4];
    void foo(const A x);
    // From this point on A behaves as if it were declared
    typedef const int A[4][4];
  ```

- In certain circumstances MrC generated a pair of loads for a long long type using the same register for the base of both of the loads and as the destination of the first load, invalidating the base for the second load.

- In some circumstances MrC did not consider array and pointer types to be equivalent. (Radar 1353590)

- MrCpp did not always correctly take into account the const and volatile attributes when determining whether or not types match. (Radar 1353592)

- MrC now reports an error for empty declarations when `-ansi strict` is in effect. The following examples will now be detected as errors.

  ```
  int ;
  int i,;
  ```

- MrCpp no longer calls operator delete if an exception is thrown during a call to the placement variant of operator new.

- MrC would sometimes generate an incorrect function prolog for functions using AltiVec and `alloca()`.

- The compiler no longer crashes while optimizing certain nested loops.

- Exception rethrows were not freeing internal memory leading to a possible premature termination of execution if enough rethrows were executed. This problem has been fixed in a new version of MrCExceptionsLib.

- Use of the `-alias type` optimization option could result in some loads and stores being removed erroneously.

- MrC would sometimes optimize expressions of the form `x-c != 0` incorrectly. At times this optimization would come into play with loops with a constant number of iterations, resulting in the last iteration being skipped. (Radar 2438831)

- In rare circumstances MrC would incorrectly optimize casts done out-of-line by library routines (e.g., unsigned int to float).

- MrC would sometimes make incorrect assumptions about the lifetime of variables in code containing large numbers of assignments. (Radar 2427563)

Bug Fixes in v. 4.1.0a8

- A problem was introduced in MrC 4.1.0a7 which would cause the compiler to crash when reporting an error when no matching declaration was found for a function call.

- Given nested structs, if the inner struct is declared with a tagless typedef ("`typedef struct {...} foo;`"), and if the outer struct has a virtual function, then MrCpp would issue an erroneous warning about an untagged nested typedef while generating the compiler-generated constructor for the outer struct. (Radar 1325743)

- The compiler did not detect assignments to array elements in const member functions when the array is also a member. It also did not detect assignments to elements of const arrays. (Radar 1213512 and 1353576)

- When exceptions were enabled MrCpp would occasionally crash when compiling functions with returns before the end of the function.

- The compiler did not detect attempts to call or take the address of `main()`. (Radar 1191083)

- MrC occasionally generated incorrect code for the AltiVec `vmsummbm()` function, reversing the first two operands. (Radar 2354545)

- Under rare circumstances MrC would generate references to stack frame locals without a stack frame having been created while attempting to delay the prolog in functions with early returns. (Radar 2355537)

- MrC was incorrectly converting the constant -0.0 to +0.0 when converting long double. The code for `(vector float)(-0.0)` was also incorrect. (Radar 1663379 and 2326138)

- Particularly complex templates could result in mangled names (function signatures) which exceeded MrCpp's 1024 char symbol name limit. The limit has not changed, but template class names are now compressed in mangled names if the name is too long. Note that new versions of the unmangle tool, dumpxcoff, unmangler library, and debuggers are needed to properly disassemble the new compressed names. All of these are forthcoming.

- MrCpp would sometimes emit an erroneous error message on declarations of objects with namespace qualification when there were preceding declarations with class qualification.

- MrCpp did not take the values of non-type template parameters into account when matching templates.

- MrCpp did not handle namespace qualification in base class specification in class declarations and in base initializers in constructor definitions.

- MrCpp would report a duplicate template function error in cases where a definition of a template is followed by a declaration of the same template.

Bug Fixes in v. 4.1.0a7

- The VRsave register was not being correctly set in functions not directly using AltiVec constructs that contained inlined calls to functions using AltiVec constructs.

- MrCpp complained about attempts to pass non-const pointers when const pointer parameters were expected.

- MrC did not insert an implicit white space between characters immediately preceding or following a macro expansion and the characters from the macro expansion itself. In other words, a token cannot be created partly from characters coming from a macro expansion and partly from characters not coming from a macro. (Radar 2330913)

- A bug was introduced recently into MrCpp which caused it to have problems with template functions which are called recursively.

- MrC would assert when attempting to cast a const float to a void (e.g., the statement "`(void)x;`" where `x` is a const float).

- MrC would occasionally (depending on the contents of memory) incorrectly generate macro expansions with the concatenation operator (`##`), leading to spurious compilation errors.

- MrCpp was generating bad calls to __vec_ctor() when exceptions were enabled. Calls to __vec_ctor() are generated when instantiating an array of objects.

- Asserts could result from the use of `#pragma outofline` and `#pragma seldom`. The code generated for `#pragma outofline` was incorrect.

- MrC was incorrectly trying to use integer registers for the operands of vector predicates when vector registers should have been used instead. (Radar 2338985)

- Use of the `__rlwimi` intrinsic could lead to asserts in compiler or bad code.

- Occasionally MrC would generate immediate constants in instructions incorrectly when the constants were used by intrinsics or AltiVec operations appearing in loops.

- The AltiVec VRsave register was not being set correctly in functions using all of the vector registers v0 through v15. The compiler was generating a LI (load immediate) of the callee's VRsave mask but not OR'ing it with the caller's mask.

- MrCpp incorrectly reported an error for the use of user-defined overloads of operator functions for scalar types.

- Occasionally MrC's optimizer would remap a register to r0 in an instruction where r0 was not permitted.

- MrCpp did not recognize valid user-supplied conversions for classes passed to AltiVec functions.,

- MrCpp did not correctly handle user-supplied overloads for AltiVec intrinsic functions.

- MrCpp did not report an error when a storage class (e.g., "static") was specified on a member function declaration.

- MrCpp was not correctly distinguishing functions with arguments that were different instances of the same template.

- MrCpp did not handle namespace-qualified symbol references in struct tags. It also did not always restrict itself to the specified namespace when resolving namespace-qualified symbol references. It also did not handle symbol references with namespace qualification followed by class qualification.

- In cases of loop induction variables which were signed chars, MrC would sometimes incorrectly remove the induction variable from the loop even though there were still uses of the induction variable in the loop.

- In cases of leaf function using AltiVec operations but having no AltiVec locals, the compiler did not correctly set up the stack frame to access non-vector locals.

- The compiler would assert if an vector constant was passed directly to an AltiVec intrinsic function.

Bug Fixes in v. 4.1.0a6

- MrCpp could not correctly parse "`sizeof(n::t)`" where `n` is a namespace and `t` is a typedef.

- In certain circumstances MrC would incorrectly optimize a loop using a load-store-update construct without generating all of the required increments.

- In certain circumstances when optimizing cases where the function prolog could be delayed beyond a test of a condition, MrC would generate incorrect code.

- An assertion in the back end of the compiler could result from a previous "fix" to a problem involving the destruction of base classes in a destructor.

- There was a problem in the fix done in 4.1.0a5 to handle matching of pointers to const with formal parameters that were const arrays.

- MrCpp would sometimes incorrectly report an error when assigning to a mutable field. It also failed to match function parameter types when an actual parameter was a mutable field.

- The recent fix for MrCpp's handling of const and volatile attributes when determining the best match for overloaded functions required further refinement.

- Certain constructs involving logical operations on string constants could result in generation of bad code or in a compiler assert.

Bug Fixes in v. 4.1.0a5

- In certain circumstances MrCpp was confusing struct/class/enum tag names with something else with the same name declared in the same scope.

- MrCpp was not taking the const and volatile attributes into account when determining the best match for overloaded functions.

- Constructs of the following form would cause the local optimizer to go into an infinite loop.

  ```
  if (<condition>)
    return 1;
  return 0;
  ```

- The CodeWarrior plug-in version of MrC continued to have problems in which warnings were treated as errors.

- In certain circumstances MrC would crash in the local optimizer while folding constants.

- In certain circumstances use of the `-ansifor` option would lead to spurious errors in function template instantiation.

- A recent fix for a problem in the destruction of base classes in destructors would occasionally result in an internal error.

- MrC[pp]'s optimizer would occasionally generate incorrect code for expressions of the form `“(x & mask) >> shift”` where the mask and shift values are constants. (Radar 2319072)

- MrC[pp] had a problem matching types of actual and formal parameters when pointers to const parameters were passed to functions with const array parameters that had been declared with typedefs.

- In certain circumstances MrC would crash or assert while optimizing loops.

Bug Fixes in v. 4.1.0a4

- MrCpp would assert with an internal error when encountering a call to a static member function if the function reference in the call was qualified with the class name and parenthesized, e.g., `(S::foo()`.

- Macro expansions in preprocessor directive lines resulted in an error when the `-ansi on` option was in effect.

- MrCpp did not generate code to adjust an object’s virtual-table pointer when destructing that object’s base classes.

- A problem was introduced in MrC[pp] 4.1.0a3 which caused warnings to be treated as errors.

Bug Fixes in v. 4.1.0a3

- In certain circumstances MrC would generate bad code for testing loop termination for loops with more than one induction variable which the compiler erroneously converted to a load-store-update form. (Radar 2290938)

- Invalid code was generated for certain loops with “artificial loop exits” where the sense of the loop condition is changed by modification of the induction variable. (Radar 2284970)

- Multiple uses of the same constant in different vector operations could lead to a compiler assertion in cases where the constant was needed both in a register and as an immediate operand in an instruction. This is a fix to a previous bug (Radar 2281875) “fix”.

- The compiler sometimes mishandled the `__eieio()` intrinsic during instruction scheduling.

- A previous improvement in instruction scheduling of vector register-to-register moves replaced some `vmr` instructions with `vsldoi` instructions in order to avoid instruction unit conflicts. This prevented the compiler from removing some unnecessary register-to-register moves.

- The compiler sometimes misscheduled AltiVec instructions involving registers V14 through V19.

- MrCpp sometimes mishandled the placement of dtor calls for compiler-generated temporaries. In some cases the dtor calls were placed at the end of the current scope rather than at the end of the current statement, allowing paths to the dtor that did not first go through the corresponding ctor.

- When generating the code for a constructor MrCpp was not able to find the constructor for a base class if the constructor for that base class was declared to be explicit.

- The alignment of structures containing arrays of vector types was not being set correctly.

- The compiler would sometimes generate a bad object file when a function which was forward declared was later defined as using AltiVec instructions.

- With exceptions enabled MrCpp generated a spurious destructor call in the constructor of classes which contained member arrays which needed constructing. The spurious destructor call would destruct the first element of the array.

- MrC generated incorrectly optimized the case of shifting a long long value left by exactly 32.

- MrCpp did not handle non-type arguments in function templates. It also did not handle the explicit template function instatiation syntax, i.e., `tfunc<args>(params)`. MrCpp also had several problems with template function overload resolution, including problems resolving overloaded template functions inside of namespaces. MrCpp sometimes mistakenly reported errors on default values for template function parameters.

- In certain cases MrC would attempt to use LSU (load-store-update) operations on objects larger than 32,767 bytes, resulting in offsets too large for the instruction encoding.

- In rare circumstances the compiler would assert with a “GRA ERROR” (an error in the global register allocator) due to a bad edit in the addition of AltiVec support to the compiler. (Radar 2305315)

Bug Fixes in v. 4.1.0a2

- An improvment was made to the fix reported below for Radar 2281875. The new fix makes further optimizations possible.

Bug Fixes in v. 4.1.0a1

- The compiler incorrectly optimized away some loops with long long interation variables and fixed bounds.

- Calls through function pointer variables sometimes resulted in random compiler crashes. (Radar 2280663)

- Loading a precompiled header file created with the 4.x series of MrCpp sometimes resulted in random compiler crashes. There was a problem in restoring the global unnamed namespace, which is always used implicitly whether namespaces are used explicitly or not.
- Incorrect code was generated in some instances for vector operations with three arguments (e.g., `vperm`) at `-opt speed` and `-opt size`. (Radar 2281673)
- Under certain circumstances a conditional block optimized to be placed before the function prologue did not set `VRsave`. (Radar 2281871)
- An internal compiler error sometimes resulted from the use of the same constant value as an argument to different vector instructions. (Radar 2281875)
- The `VRsave` register was not set correctly for forwardly declared static functions using vector operations.
- Negative long and long long constants were being converted to unsigned constants with the same effective value. This could result in problems when the constant was cast to a floating point type.

Bug Fixes in v. 4.1.0d4

- The pointer returned by the placement form of vector new was off by four bytes and would result in an error when passed to vector delete.
- The warnings produced by the `warn_uninit` and `warn_maybe_uninit` modifiers to `-opt speed` and `-opt size` were not put out by the compiler's normal warning mechanism (meaning, for instance, that these warnings could not be turned into errors using the `-w iserror` command line option).
- MrCpp did not recognize typedef names for classes in the syntax for base class initializers. (Radar 225236)
- A number of internal problems involving long longs have been fixed. Some of these were internal bookkeeping errors resulting from a failure to take into account that a single long long takes two registers. (Radar 2255162)
- An internal compiler error would result from macro definitions using the "##" concatenation operator where the left hand argument to the operator was a macro parameter with an ordinal position corresponding to the ASCII representation of a space or tab character.
- There was a 16000 character limit on the size of any macro parameter.

- Some expressions involving right shifts by constants were incorrectly optimized away when they appeared in conditionals. (Radar 2269651)

- Warnings were not issued for shifts by constants greater than 32.

- The code generated for expressions involving signed short or char values involving addition, subtraction, or multiplication by a constant was less than optimal. (Radar 2269652)

- The -opt speed optimization for division by a constant (other than a constant which is a power of two) generated incorrect code. This problem was introduced in the 4.1 compilers. (Radar 2274418)

- Division by the constant -1 did not correctly set the sign of the result. Division by 1 and -1 are handled as a special case by the compiler, however the negation of the numerator was overlooked in the -1 case.

- A bug was introduced in version 4.1.0d3 of MrCpp in a fix for the problem where code generated for calls to `dynamic_cast` would try to dereference its first argument without checking whether it was NULL. Under certain circumstances the bug that was introduced would result in infinite recursion in the compiler and a resulting crash.

- Long long multiplications were not correctly handled correctly in some cases where the operand and result registers were the same.

- The compiler sometimes produced an internal error when processing an operation involving a move from a location specified by the address of a parameter. The following are example of such constructs:
  ```
        aggregate_value = *(Aggregate_type *) &parameter;
        x = vec_lvsl(0, &parameter);
  ```
  (Radar 2268454)

Bug Fixes in v. 4.1.0d3

- There were a large number of problems reported against MrC's preprocessor. The preprocessor was re-written in order to fix all known problems. Specific problems fixed include spurious blanks introduced by macros intended to generate identifiers, macros which behave differently on successive invocations with the same arguments, failure to stop the expansion of calls to macros already being expanded, over-evaluation of some arguments, and infinite recursion of the macro processor. Please let us know if any new problems have been inadvertently introduced in the new implementation of the preprocessor. (Radar 1307014, 1315304)

- In certain cases the optimizer would go into infinite recursion, ultimately resulting in the compiler crashing. The circumstances under which this would happen are not easy to characterize, but they involved constructs that the compiler was trying to turn into ternary expressions ("`… ? … : …`").
- A number of internal errors in the code generator and optimizer have been fixed. The circumstances under which some of these errors occur are not easy to characterize. Particular errors that have been fixed include ones which produce the following message:
  - `Register fp14 not found in list (RemFromRegList)`
- In certain circumstances, when compiling with the -opt speed or -opt size options, MrC would generate bad code for evaluating whether to branch on a conditional statement. The bug appeared in very large functions.
- The compiler generated bad epilogs under the `-opt size` option for routines using enough floating registers to require saving and restoring non-volatile floating point registers.
- MrCpp did not try operator conversions when encountering an object as an array subscript. (Radar 1221592)
- The compiler issued "variable not used" warnings for some compiler-generated temporaries with names of the form "`.TMP…`".
- Under the `-opt speed` and `-opt size` options the compiler generated bad immediate values for some `vsldoi` instructions.
- MrC did not handle the scheduling of vector saturate instructions correctly. Also, under certain circumstances, the ordering of `mfvscr` and `mtvscr` instructions could be incorrect.
- MrC switched two of the arguments to the `vmaddfp` and the `vfnmsubfp` vector functions. (Radar 2258445)
- There were a number of other problems which appeared in MrC[pp] 4.1.0d2 when used in conjunction with the new StdCLib released at that time, including the creation of bad object files. These problems were the result of an error in the `realloc` function in that library.

Bug Fixes in v. 4.1.0d2

- A number of code generation problems have been fixed. Many of these bugs only appeared under subtle combinations of conditions which are not easily described without referring to the internal operation of the compiler, so there isn't much to say about them other than that we're happy we found them and fixed them.

- Given a class A with an explicit virtual destructor with an explicit empty exception specification, and given a class B inheriting from A, the compiler automatically generates a virtual destructor for B. In doing do, the compiler did not bother to propagate the exception specification from the destructor in class A, but then issued an error for a mismatched exception specification. The appropriate exception specification is now propagated. An instance of this problem appears in the stdexcept.h header.

- The compiler did not always propagate the correct type information when reducing expressions of the form "`*(cast)&var`" to "`var`", so an expression like "`*(ULONG *)&buf[4]`" would result in memory accesses to the wrong location. (Radar 1655090)

- The compiler would sometimes dereference pointers on the right hand side of question-colon expressions before the left hand side was fully evaluated. (Radar 1669841)

- In certain circumstances involving user-defined conversions involving enums the compiler would generate a spurious ambiguous conversion error. (Radar 1631487)

- In power alignment mode certain cases of structures whose first element was a declaration of a nested structure or union containing a double would have a different size when compiled by MrCpp and by MrC. (Radar 1677141)

- In certain cases the optimizer would use incorrect registers in load/store update instructions accessing loop induction variables. (Radar 1647531)

- The produced a bad fixup entry in the object file in cases where the only reference to a destructor was from an exception table entry for a compiler-generated constructor.

- The compiler mistakenly typed constants of the form "`<n>e<m>f`" as doubles instead of floats.

- The compiler mistakenly made `sizeof(+x)` equal to `sizeof(x)` instead of `sizeof(int)`.

- The optimizer erroneously turned a loop's conditional branch into an unconditional branch in certain conditions when the initial value of the loop induction variable was set before this loop and then the value of expression expressing that initial value was also changed before entering the loop. (Radar 2219054)

- Destructors that were called through thunks (which adjusted the this pointer) were not using the adjusted this pointer for the operator delete call that followed. (Radar 1172105)

- The digraphs for the #, ##, {, }, [,] were not recognized.

Known C++ Language Deficiencies

- **Templates**. Partial template specializations are not supported. Partial ordering of function templates is not supported.

- **Standard libraries**. There is a version of the STL library for MrC available from STLport. The stream headers and libraries are missing some classes. The headers and libraries do not use namespace std.

- While overloading of operator new and delete is supported, overloading of operator new[] and delete[] is not.

- Operator new and typeid expressions do not throw exceptions.

Known Problems

- MrC's Commando dialogs do not support all of the latest changes in command line options.

Manual Errata

- Page 3-52 of the "MrC/MrCpp: C/C++ Compiler for Power Macintosh" manual incorrectly states that the "__option(<option_name>)" feature can be used in normal conditional statements. Its use is actually limited to preprocessor conditionals. The following shows the correct way to test whether the ansi option is set (as opposed to what is shown in the manual).

```
#if __option(ansi)
...
#endif
```