MPW QR4 C
Release Notes

The primary changes that have been made to the C compiler for MPW QR4 are for the purpose of optimizing code size and performance. Other important changes are:

- An option to control the level of optimization.
- An option to control the level of warning messages.
- An option for machines having the MC68020 and up that allows stand-alone code segments greater than 32K.
- An option for invoking the "32-bit everything" run-time architecture.
- Provision for passing parameters to inline functions via registers by using `#pragma parameter`.
- Support for the MacApp debugger,and code profilers, controlled by a command line option and a `#pragma`.
- A pragma to prevent in an efficient manner the multiple inclusion of header files.
- A pragma to force the generation of MC68020 code.
- Prevention of dead code stripping when linking by use of `#pragma force_active`.
- Several other pragmas comparable to or related to existing command-line options.

-opt

This option controls the application of optimizations to the generated code. The syntax is:

`-opt` *param-list*

where

param-list = *param*[,*param*]

param = *exclusive* | *nonexclusive*

exclusive = on | off | full # choose one at most

nonexclusive = nopeep | nocse # choose zero or more

The meanings are as follows:

`off`	no optimizations
`on`	“normal” optimizations
`full`	include register allocation
`nopeep`	inhibit peephole optimization
`nocse`	inhibit the elimination of common subexpressions

The default is `-opt on`. The choice `-opt full` causes any free registers in the temporary storage group (A0-A2, D0-D3) to be used for the storage of variables. This choice increases compile time and may or may not significantly reduce execution time.

-warnings

The following command-line options replace the former options `-w` and `-w2`:

`-warnings on`	normal warning level, the default
`-warnings full`	maximum warning level
`-warning off`	no warnings

-bigseg

The `-bigseg` command-line option allows code segments to grow beyond the 32K limit. It uses the 68020 PC-relative modes of addressing with 32 bit offsets. This obviously limits code produced with this option to machines with a 68020 or higher numbered CPU. The advantage is that it can produce code which is entirely contained within a single large segment and which requires no runtime overhead of address fixup (possibly not even a jump table). This can be especially useful in porting UNIX tools written for systems which know no artificial limits on code size or for developing large Macintosh CDEVs or desk accessories. The only other requirement for these code resources is that all externally defined entries must be within the first 32K of code.

-model

The following options have been added to support "32-bit everything," the run-time architecture that removes the 32K restriction on segment size, jump table size, and the size of the global data area:

`-model near`	the default
`-model far`	remove the 32K restriction for both code and data
`-model nearData`	the default
`-model farData`	remove the 32K restriction for data
`-model nearCode`	the default
`-model farCode`	remove the 32K restriction for code

For further details, see MPW QR4 Run-Time Architecture Release Notes.

#pragma parameter

This is used to specify that parameters for inline functions be passed directly in registers. The syntax is shown below:

Reg ::= __D0 | __D1 | __D2 | __A0 | __A1

```
#pragma parameter [Reg] functionname [(Reg[,Reg])]
```

It is important to note that the only registers that may be specified are `D0`, `D1`, `D2`, `A0`, and `A1`. These registers are considered to be temporary registers used internally by the C compiler. Thus, if any function is using those registers and then makes another function call, the calling function will generate save and restores of the affected registers around the call. This knowledge allows the safe use of those registers within the inline function.

The `#pragma parameter` directive must precede the inline function declaration, designated by `functionname`, to which it applies. The size and order of parameters is determined by the function prototype alone. The pragma then directs those parameters to be placed within the specified registers. The first register, which optionally may be omitted, determines where the function result will be returned. Next appears the required function name. It must agree with the name of the inline function definition to which it applies. This is optionally followed by a list of registers enclosed within parenthesis. This list contains one or more registers separated by commas. The leftmost function parameter will be assigned to the first register listed in the parenthesis. The remaining parameters will be assigned in order to the remaining listed registers. The number of registers requested need not match the number of parameters in the function prototype; extra listed registers will be ignored. When there are more parameters than specified registers, the extra parameters will be placed on the stack as they normally would have been in the absence of the `#pragma parameter` directive.

Other than the actual placement in registers, these parameters will be assigned in accordance with the either the normal C or Pascal parameter-passing conventions. That is: Pascal style functions will assign parameters of the size specified, from left to right, first to registers and then pushing the remainder on the stack. C style parameters will always be promoted to size int and then placed on the stack from right to left until the remaining parameters match the number of specified registers. Those leftmost parameters will then be placed, also as ints, in the correspondingly ordered set of registers.

The function return will also be handled by normal conventions: Functions returning type void will have no return regardless of whether or not an initial register was specified. For other return types with a specified return register, the Pascal statements will move correct-sized results from the register, while those returned from standard C functions will be of size int. If there is no initial register specified, the default rules will apply: Pascal style routines will return appropriately sized function results from the stack, while C style routines will return int sized values in D0.

This pragma can be used in the C interfaces to allow some Macintosh calls to be rewritten as inline functions requiring no glue. An example would be:

```
#pragma parameter __A0 NewHandle (__D0)
pascal Handle NewHandle (Size logicalSize) = 0xA122;
```

The compiler will generate code to move logicalSize directly into D0. The entire body of the function will be the inline trap $A122. Finally, the result will be moved directly from A0 to the assignment variable. Rewriting these interfaces will result in smaller, faster code when compared to the glue which it would be eliminating.

-trace and #pragma trace

`-trace` and `#pragma trace` will turn on support for the MacApp debugger or for a code profiler to give performance information. When this feature is on, the C compiler will insert a function preamble and postamble into the code. The preamble will be: `JSR %_BP` inserted after the `LINK` instruction on entry into the function. The postamble is: `JSR %_EP` inserted just before the `UNLK` instruction on function exit. The `%_BP` and `%_EP` calls, which take no parameters, must then be provided by the programmer to support either MacApp debugging or profiler support.

The "`#pragma trace on|off`" option is placed in source code to turn this feature on or off for an individual function only. The preferred location for the `#pragma` would be immediately before the function definition. However, it can be recognized while in the body of the function itself.

- Note that all pragmas relating to a function are in effect collected and processed before the code of the function is processed, so that only that last declaration of `trace on` or `trace off` for a given function will be honored.

The command line option "`-trace on|off|always|never`" provides more control. The `on` and `off` options provide the default setting, which can then be overridden by `#pragma trace` statements embedded within the source code. Choosing `always` or `never` overrides the #pragma settings, respectively forcing all functions to either have the function pre- and post-ambles or not.

#pragma once

The #include "once" feature provides an efficient way of inhibiting multiple includes of the same file. We currently handle this by inserting a sequence of the form:

```
#ifndef name
#define name
...
#endif
```

inside of an include file to suppress inclusion of the main body of the file more than once. This, however, requires the compiler to open the file *each* time (applying the appropriate search rules), and to scan through the entire include file looking for the delimiting `#endif`. This time consuming process can be considerably reduced by providing a mechanism whereby the compiler maintains a list of the files that have been #included and takes the responsibility of inhibiting multiple inclusion.

The "once" feature is assumed by the compiler recognizing either of the following conditions in a #include file:

- The presence of `#pragma once`. This unconditionally indicates the file is to be included only once.
- The presence of an initial `#ifdef`, `#ifndef`, or `#if`, and the delimiting `#endif` surrounding an include file (ignoring comments). Since there may be times when a user would *not* want to have the compiler automatically suppress multiple inclusion, the option `-notonce` has been provided. However, if the user has explicitly inserted `#pragma once` in a file, the `-notonce` option is overridden for that file.

The advantage of the pattern recognizer is that the use of conditionals surrounding the contents of include files to suppress reprocessing is a common practice. Indeed, it is what is presently done in all our CIncludes with the exception of Assert.h. Thus, no changes to our CIncludes need to be made in order to use the "once" enhancement.

The #pragma complements the pattern recognizer. It can always be used. Also, it is compatible with some other existing C compilers (e.g., GNU GCC). The word "once" was chosen to attain this compatibility.

#pragma processor

The choice between generating 68000 code and 68020 code can be made by use of the pragma

```
#pragma processor 68000|68020.
```

The default, of course, is to generate 68000 code.

#pragma force_active

The syntax of this pragma is

```
#pragma force_active on|off
```

The default is "off." The effect of the pragma with the parameter "on" is to prevent the Linker from deleting the code within the scope of the pragma in the event that there are no references to it, an action known as "dead code stripping."

#pragma push, #pragma pop

This pair of pragmas saves (`#pragma push`) and restores (`#pragma pop`) the conditions set by the options (or pragmas) `-s`, `force_active`, `-mbg`, `-warnings`, `processor` (or `-mc68020`), and `-opt`.

mbg, warnings, and opt

The following three pragmas correspond exactly to the options of the same names:

```
#pragma mbg off|full|on|ch8|0…255
mbg ch8        # v2.0 compatible macsbug symbols
mbg off        # no macsbug symbols in the code
mbg on|full    # full macsbug symbols
mbg <n>        # macsbug symbols to length <n> (<n> can be 0..255)

#pragma warnings on|off|full  # set warning level; "on" is the default

#pragma opt off|on|full
opt off        # don't apply code optimizations
opt on|full    # choose level of code optimization (on is default);
               #  can modify with [,nopeep] [,nocse]
```

```
# (no peephole, no common subexpression)
```

Linking Requirement for 881

△ **Important** The library CLib881.o has a new positioning requirement as a consequence of the elimination of CRuntime.o. CLib881.o must now not only precede all members of {CLibraries} in the link sequence but must also precede the library Runtime.o. △

Changes to C Library and Interfaces

Libraries

ANSI C

The MPW QR4 C libraries are largely in conformance with the current ANSI C standard (X3.159—1989).

File Types

If you create a new file and do not specify that it is a binary file by including `'b'` as part of the open mode of an `fopen()` call or `F_BINARY` as part of the mode for an `open()` call, the file that is created will be of type TEXT. Except for marking a file as a TEXT file at its creation, there is no difference in the way that MPW C handles text and binary files internally.

There is also a new function which can be used to set the creator and file type of any file. The function is:

```
void fsetfileinfo (char *filename, OSType newcreator,
                                   OSType newtype);
```

cfree()

Because `cfree()` is not part of the ANSI standard this function has not been supported since the release of MPW 3.0. The code for `cfree()` has now been removed from the libraries. If you have any existing code which uses this function, you can create your own macro or function to replace it. Since MPW C has always ignored the second two arguments of `cfree()`, the function call `free(p)` is equivalent to `cfree(p, int1, int2)`.

dup()

Because `dup()` is not part of the ANSI standard and because its functionality is duplicated in the `fcntl()` function, the `dup()` function is no longer supported and will be removed from a future version of `StdClib.o` in the MPW C Libraries. The function call `fcntl(filedes, F_DUPFD, 0)` is exactly equivalent to the call `dup(filedes)`. Existing code should either be changed to call `fcntl()` directly or a macro or local definition of `dup()` should be provided by the user.

fputs()

This function will now set *errno* when applied to a read-only file.

malloc()

`malloc` is currently limited to a maximum size of about 8 MB per call. Size parameters of value 2^{23} or greater result in a null pointer being returned. This is not a new limitation, but it was not previously documented.

Interfaces

String Pointer Parameters

The definition `typedef const unsigned char *ConstStr255Param` has been introduced in types.h for compilers that support `const`. It is intended to replace the use of `const Str255`. For consistency, similar declarations have been included for `ConstStr63Param`, `ConstStr32Param`, `ConstStr31Param`, `ConstStr27Param`, and `ConstStr15Param`.

Generic Pointers

In function declarations, all instances of `Ptr` as an argument type have been changed to `void *`. This removes the restriction that parameters represented by pointers had to be of type `char *`. It is intended in the near future to make the equivalent change for Pascal by using `UNIV Ptr`.

Fcntl.h

Definitions for the following "internal-use only" macros have been removed from the header file FCntl.h.

```
O_TMP
O_USEP
```

IoCtl.h

The definition of the internal struct type `_SeekType` has been removed from IOCtl.h.

Time.h

`#define CLK_TCK 60` has been changed to `#define CLOCKS_PER_SEC 60`.

The type cast in the `#define` for `difftime` has been changed to `long double`.

Types.h

The type `Str32` has been moved from AppleTalk.h to Types.h**.**

Known Outstanding Bugs

Extra link file

When pointers are subtracted, we do a divide of the difference by the size of structs being pointed to in order to return the number of elements between the pointers. Since we are now forcing all of our arithmetic to go through int sized operators, this divide becomes a long divide. No longer fitting the DIV.W instruction in the 68000, we call a library

routine: LDIVT. You now will be required to link against CRuntime.o, even if your code did not previously need to. Sorry, this is another one we'll correct.

Initializer Type Checking

Type compatibility is not checked to the same extent for initializers as it is for ordinary assignments. If there is any doubt about the correctness of an initializer, it is safer to write the declaration without an initializer and to use a subsequent assignment statement to perform the initialization.

Incomplete Enum Declaration

Although it is legal C syntax to declare an enum with no constant values, MPW QR4 C treats such a declaration as an incomplete type specifier and returns an error message. Be sure to declare *enums* fully. This is reported as a bug inasmuch as it is a deviation from the ANSI Standard.

Assignment from Floating Point Arrays to Integer Types

When a floating point number from an array is assigned to an integer type, a library routine converts the floating point number to an integer and returns it in register D0. Unfortunately, a previously calculated array offset is placed in D0 before the actual assignment can be made.

The work-around is to assign the variable from the array to an intermediate non-array variable of the same type, and then to assign the intermediate variable to the one of integer type.

Example:

```
extended e, f[3];
 int i, j;

 i = f[j];   /* generates incorrect code */

 e = f[j];  /* generates correct code */
 i = e;
```

Assignment from Fields of Structs

If an assignment is made of a field of a struct being returned, the compiler cannot properly apply the field's offset and may assign the wrong portion of the struct.

The work-around is to assign the returned struct to an intermediate variable of the same type. Then assign the proper field from that intermediate copy to the desired variable.

Example:

```
Point GetPoint(void);

main()
{
      Point pt;
      short y;

      y = GetPoint().v; /* generates incorrect code */

      pt = GetPoint();  /* generates correct code */
      y = pt.v;
}
```

Pascal-Style Functions Returning Structs

When a Pascal-style function returning a struct is used in a complex expression, the offset to the struct may be miscalculated.

The work-around is to return the struct directly to an intermediate variable, and then to use that variable in the desired expression.

Example:

```
pascal Rect PGetRect(void);

main()
{
      Rect r, *pr;

      pr = (r = PGetRect(), &r);    /* generates incorrect code */

      r = PGetRect();               /* generates correct code */
      pr = &r;
}
```

Floating Point Parameters in Pascal-Style Functions

Pascal-Style C functions may corrupt the value of floating point parameters. The best advice at present is to use only type *extended* as parameters in Pascal-style functions. This is because the bug appears to be in the process of conversion to extended, which is the type used for floating point arithmetic.

Example:

```
pascal void FTest(float kp)
{
      printf("%f\n", kp);     /* usually prints wrong value */
}
```

-model

The code to distinguish the cases `-model farData` and `-model farCode` is not yet in operation. The use of either of the above options has the same effect as using `-model far`.

#pragma parameter

The use of complex expressions as parameters may cause a fatal compiler error with the message: "Expression too complicated."

Large data

When compiling with the -m option (to address data greater than 32k), the code generator emits code that calculates a variable's address by copying A5 into another A register and then adding the appropriate offset. In expressions involving multiple array indices, the code generator may not have a spare A register to use for the address calculation. In these cases the code generator will abort with a message like:

```
### Code Generator Fatal Error
### Register 8
### Within f (Error 2001): Expression too complicated, code
                         generator out of registers
```

We have reduced the frequency of these aborts, but they do still occur. If this happens to you, try to identify and simplify the offending expression.

Pascal Keyword

The pascal keyword is broken in the specific situation where one attempts to call a C function which returns a pointer to a pascal-style function. The C compiler misinterprets the C function as a pascal-style function and the function result is lost.

Declaration of void types

The compiler incorrectly allows variables of type `void` to be declared. (Variables of type `void *` are ok.)

Incomplete arrays

The compiler allows incomplete arrays to be declared as variables and within struct definitions. Not only is this incorrect, but relying upon it and then indexing off the result can yield unpredictable results.

Floating Point

Floating point arithmetic is handled separately from integer arithmetic and still suffers from some of the same code generation problems that have already been fixed in the integer cases. Pascal functions of type extended or float do not return the correct value.

Unsigned Constants

Any unsigned int or unsigned long constant above `0xFFFF7FFF` (which is declared either by a trailing `u` on the number or an explicit cast to `unsigned`) is treated as a signed constant when used in an arithmetic expression. This means, for instance, that dividing by such a constant will be done using a signed division operation instead of an unsigned division.

The following code illustrates the problem and shows the workaround:

```
#define UINT_MAX 0xFFFFFFFFu
main()
{
      int p, x;
      x = 2;
      p = UINT_MAX / x; /* Wrong result: p = 0 */
      p = UINT_MAX / (unsigned) x; /* Right result: p = 0x7FFFFFFF */
}
```

The Macro Preprocessor

The macro preprocessor is not a complete ANSI implementation in the sense that the rules for rescanning and replacement in the context of ## are not observed. For example:

```
#define glue(a,b) a ## b
#define HIGHLOW "hello"
#define LOW LOW", world"
glue(HIGH, LOW)
```

results in: "hello"", world"; rather than the correct: "hello".

String concatenation doesn't work when the second string is the result of expanding a macro. For example:

```
#define REASON "I felt like it"
char *message = "Program aborted because " REASON;
```

results in an error at the token REASON.

Circular macro definitions will hang or crash the compiler.

`#include <FileName>` works incorrectly. If `FileName` isn't found in the standard places (`{CIncludes}` and directories specified as searchpaths on the command line), the compiler will search the current directory for it.

`#include ":dir:FileName"` looks for `:dir` relative to the current directory, and not relative to the directory of the file containing the `#include`.

Many reports of macro expansion errors, other than the ones just mentioned, have so far turned out to have been code generation errors produced by the expanded source code. Macros are good at expanding to more complex code which will require temporary values to be stored on the stack; this is just the area of code generation that we are most likely to have problems with.

C Language Library and Interface

In `printf()`, the "`0`" character flag works incorrectly for the "f" (floating point) conversion specifier. It will NOT pad with leading zeros.

Bug Fixes

The items listed below are bugs that existed in MPW 3.0 or MPW 3.1. The list does not include bugs that first appeared in Alpha or Beta versions of MPW 3.2.

- The ANSI error directive, #error, used to work in contexts like

  ```
  #error "Ugh, this code is terrible"
  ```

  but not if you merely said

  ```
  #error
  ```

  Both forms now work. In the future they will trigger a fatal error condition which stops compilation immediately; for now, they will report the error and allow the compiler to hunt for more errors.

- The following two fixes were to facilitate compilations from C++ source: the maximum number of nesting levels was increased to 48 and the maximum length of identifiers was increased to 250 characters.

- The keyword `entry` was removed to conform with the ANSI specification.

- Conditionals such as (anyVar < 1) or (i == y) are now handled properly when assigned directly to fields within a structure. The specific bug reports we had were for assigning to a field that is either a bitfield or an indexed structure e.g.:

  ```
  s[i].f2 = (anyVar < 1);
  ```

Other Useful Information

The following properties of MPW C have caused problems for users who were unaware of them. As far as we know, they are consistent with the ANSI Standard.

"Old C" Style Function Declarations

Parameter checking (at function call) is done only for function definitions that use the prototype form:

```
int foo(int parm1, int parm2)
```

and not for definitions using the "old C" style:

```
int foo(parm1, parm2)
int parm1;
int parm2;
```

In addition, if prototype declarations and "old style" definitions are mixed, no parameter checking will be done beyond the "old style" definition.

Pascal Keyword

The "pascal" keyword for function declarations in MPW C does not do everything that the Pascal compiler does for its function declarations. The pascal keyword in function declarations results in the name being passed to the linker in all upper case letters, parameters are reversed on the stack from the normal C manner, and they are placed there in the size actually used rather than int. The Pascal compiler additionally passes parameters that are records greater than 4 bytes long as pointers to the record. For C parameters (even in functions declared to be pascal style), entire structs can be passed on the stack. Therefore, to communicate with real Pascal functions, such a parameter must be explicitly declared as a pointer to the struct. Return values that are records greater than 4 bytes are a harder problem to handle. Not only will the Pascal compiler return a pointer to the record, but it will automatically allocate room for the actual returned record. No matter if we declare the C function to return a struct or a pointer to the struct, the calling routine will not automatically provide storage for it. Thus, such pascal type functions cannot be created in C without providing some type of assembly language glue. This was a design decision made in the early days of MPW C and is currently under review.

Load/Dump

Only one `#pragma load` directive is allowed per compiled unit.

Note that `#pragma dump` saves only symbol information. Since initializers actually generate code, we cannot load statements such as:

```
static const Boolean done = false;
```

Character Comparison

The user should be aware that character literals, e.g. `'∂'` and elements of string literals, e.g. `a[1]`, where char a[] = `"a∂b"`, are signed quantities, and will be sign extended before a (long) compare to a char variable is performed.

Contradictory statements to be found in earlier MPW C Release Notes are erroneous and should be ignored.

//-Style Comments and Backslash-Carriage Return

Macro definitions can continue over multiple lines by escaping the carriage return with a backslash character. //-style comments run from the // characters to the end of the line. So what should the behavior be when a backslash-CR combination appears at the end of a line containing a // comment? This turns out to be an incredibly controversial issue, with strong arguments on both the "Continue the comment" and "Terminate the comment" sides. So be warned: the behavior is, at least for now, officially undefined and subject to change in future releases.

Unterminated Strings as Macro Text

Macros of the form

```
#define UNCLOSED_STRING "There's no closing quote
```

or

```
#define UNCLOSED_CHAR 'x
```

are automatically closed at the end of the line; the single or double quote, as is appropriate, is implicitly added.