Porting GNU C Programs to Metrowerks' CodeWarrior C Compiler

A Metrowerks White Paper

By Tom Thompson

Porting GNU C Programs to Metrowerks' CodeWarrior C Compiler

The inevitable differences in language syntax and library implementations between these two compilers are summarized here. This information can guide programmers in adapting GNU C programs so that they build successfully with the CodeWarrior compiler.

By Tom Thompson

IF YOU'RE READING THIS, you're either seriously considering moving your software development work to Metrowerks' CodeWarrior, or have just adopted the CodeWarrior tools (congratulations!). In both cases, you undoubtedly own a body of proven C programs that were written with a GNU C compiler. And, you're wondering how difficult it's going to be to migrate such code to the CodeWarrior compilers.

Nothing strikes fear into the hearts of seasoned programmers more than the oft-quoted vendor phrase of "a simple recompile is all it takes." Because each C/C++ compiler seems to implement its own dialect of the language, you must often modify the code so that it's palatable to the other compiler. In short, that "simple recompile" can sometimes become a major programming effort. Occasionally, the amount of work spent porting the code is large enough to make you wonder if it would have been easier writing the program from scratch.

Let's be honest here: you *will* have to revise a GNU C program so that it works with the CodeWarrior C compiler. How much work this involves will depend upon many factors. This paper describes some of the differences between the two compilers in terms of conformance to the C language standard and library support. This information should help you determine the amount of effort required to port the source code to CodeWarrior.

There are two important facts to note before we proceed further. First, do not construe this information as a criticism of the GNU compilers. GNU C compilers have produced hundreds of industrial-strength programs, and the GNU edition of CodeWarrior for Linux uses GNU compilers.

Second, the information in this paper is neither exhaustive nor complete. There are many versions of GNU C compilers, and each version can run on several operating systems. CodeWarrior also supports a variety of processors and platforms, and while Metrowerks strives to keep the implementations consistent, there can be minor differences among them. Because each compiler implementation and set of system libraries can affect the porting process in unique ways, a single paper can't cover all of the possible variations. What this paper can do is provide some general guidelines that will serve as the starting point in the porting effort.

Getting to There from Here

Many of the problems encountered in porting GNU C source code can be broken out into three broad categories. These are implementation issues, language variations, and library issues.

- *Implementation issues*. These problems can also be considered architectural issues, because many of the compatibility problems in this category are the result of differences in the underlying hardware. While desktop programmers rarely have to worry about the hardware of their target platform changing drastically, the situation is different for embedded programming.

  Rewriting code costs money, and it opens the possibility of introducing new bugs to stable code. This makes embedded application designers quite averse to revising code—if they can help it. However, to meet a new embedded product's specific performance goal or price point, a hardware design might require a different processor. Result: a code port is necessary. Since GNU and CodeWarrior compilers support a variety embedded processors, code ports are likely between the two, and so these hardware issues surface.

  Another culprit in this category is the interfaces to system libraries. These may differ slightly, or demand a particular data alignment.
- *Language variations or dialect issues*. The source and target compilers might interpret a C program differently, or be based on a different standard. Or, the two compilers might be out of step in supporting features found in the latest standard. Whatever the reason, this creates problems in the source code where the compilers diverge. Others problems arise through the use of language extensions or features provided by the source compiler, but aren't present on the target compiler.
- *Library issues*. Many GNU compilers allow the programmer to tap into the services provided by the host machine's system libraries. Since GNU began life on UNIX-hosted workstations, the GNU C library provides interfaces to BSD, POSIX, System V, and other UNIX-style operating systems.

  Porting a GNU program to CodeWarrior presents some challenges because of CodeWarrior's roots as a desktop compiler. The Metrowerks Standard Library (MSL) implements the standard C library services, plus a subset of UNIX calls. If your program relies heavily on a particular set of UNIX services (say, BSD), you'll need to

revise the code to work within the set of services and UNIX calls that the MSL offers, or use some of the platform-specific APIs to provide the missing services.

With our problem road map firmly in hand, let the trouble-shooting tour begin.

Implementation Issues

Many of these problems are brought about by changes in the underlying hardware, not the compiler. A new processor or platform can respond differently to the same basic program operations. Such changes require a rewrite of those functions that expect a specific result based on the old hardware's behavior. Because of the diversity of processors/platforms that both the GNU and CodeWarrior compilers support, implementation issues brought about by hardware changes are common in porting from GNU to CodeWarrior.

Having said that, be aware that implementation problems aren't limited to hardware changes alone. Subtle changes in library interfaces or how a compiler sizes a fundamental data type, such as `int`, can create havoc as well.

Endian Differences

If your code migration involves using the same processor, then hardware endianess won't affect the code port, and you can skip this subsection. However, if the code port also requires moving to a processor, then you've got work ahead of you.

Disparate processors organize data in memory in different ways. One such arrangement is termed *big-endian*, which stores a multi-byte data element's most significant byte (MSB) in lowest address, while its less significant bytes occupy higher consecutive addresses. The name big-endian arises for this scheme because the "big" end of the element appears first in physical memory.

A second memory scheme places the data element's least significant byte (LSB) in the lowest address, while the more significant bytes fill the higher addresses. With the "little" end of the element appearing first in physical memory, this arrangement is called *little-endian*.

These different memory arrangements are characteristic of a processor's *endianess*. An alternate—and more descriptive—moniker, *byte-ordering*, is also applied these memory schemes. Processors in the PowerPC, 68K, and MIPS families use big-endian byte ordering, while the x86 family uses little-endian ordering. (By default, the PowerPC and MIPS processors operate in the big-endian mode; they can be programmed to operate in the little-endian mode.) There are no technical or performance advantages to either byte-ordering scheme; the differences are arbitrary and seem to exist only to confound programmers.

Figure 1 illustrates where an example data structure's elements reside in physical memory, depending upon a processor's byte-ordering scheme. One thing to note is that the bytes that comprise the character array `mess` occupy the same memory locations, regardless of the processor's endianess. It's only in elements larger than a byte where the endianess affects the placement of the data.

Sample code fragment

```
struct {
   char * cp;        /* 0x11121314                  32-bit pointer  */
   char   mess [7];  /* 'm','e','s','s','a','g','e' byte array      */
   short  flags;     /* 0x3132                      16-bit variable */
   int    header;    /* 0x41424344                  32-bit variable */
} theStruct;
```

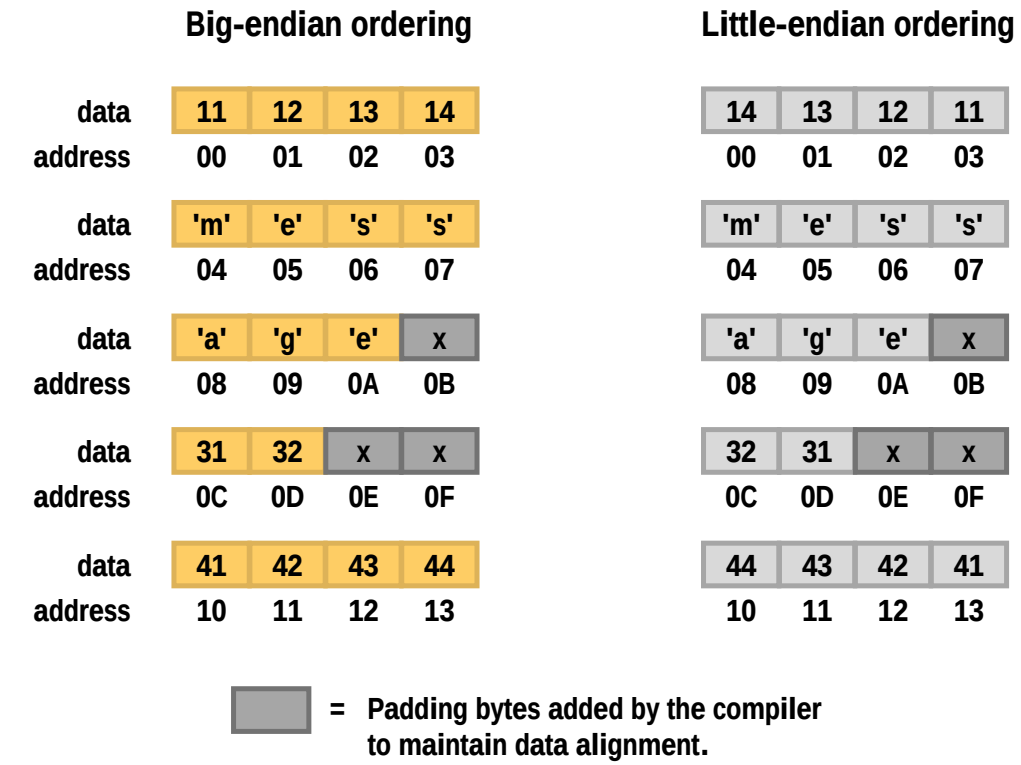

Figure 1: The two possible byte-ordering schemes for a data structure in memory.

As long as you reference items in a data structure by their respective data type, a change in the byte-ordering scheme shouldn't create trouble. It's when you probe a multi-byte data element's internals for information that the problems arise. For example, suppose

you're using `theStruct` from Figure 1, and you plan to read a byte within the `flags` variable (perhaps to extract a version number). If the byte-ordering changes, you wind up accessing the wrong byte, and the program reaches the wrong conclusion. Another problem area is when you use a `union` to build buffers that store data as one type, and retrieve data from this buffer as a different type.

The following code fragment illustrates both of these problems:

```
#define VERSIONMASK 0x0F

union {
short headerFlags;
unsigned char flagByte[2];
} flagsBuff;

flagsBuff.headerFlags = theStruct.flags;
/* Gets wrong value if byte-ordering changes */
versionNumber = flagsBuff.flagByte[0] & VERSIONMASK;
```

One possible fix for this code fragment would be to use a preprocessor directive that makes the code portable:

```
#ifdef INTEL
#define ENDIAN 0
#else
#define ENDIAN 1
#endif

#define VERSIONMASK 0x0F

union {
short headerFlags;
unsigned char flagByte[2];
} flagsBuff;

flagsBuff.headerFlags = theStruct.flags;
/* Directive fixes byte-ordering */
versionNumber = flagsBuff.flagByte[ENDIAN ? 0 : 1] &
VERSIONMASK;
```

A good way to locate this sort of problem is to examine all pointer and array references that use a fixed offset value, such as the code fragment's use of `flagsBuff.flagsByte[0]`.

Embedded programs suffer the most from a change in byte-ordering. This is because they rely on a specific byte order as the code manipulates bits within a 16- or 32-bit data element. Such elements often represent memory-mapped control registers, whose individual bits toggle signal lines that operate power relays or other devices.

Simply put, such bitfield operations aren't portable. Every bit manipulation function must be retooled, as do any control logic tests that use a mask and Boolean operations to test or set particular bits. If you're wondering whether `VERSIONMASK` in the code example requires fixing because of a change in byte-ordering, recall that it works on just a single byte.

The situation may not be as bad as it appears if the code has come from a little-endian processor and the target CPU is a PowerPC or MIPS processor. They can be programmed to operate in little-endian mode—assuming that the device's peripheral hardware will operate in little-endian mode. CodeWarrior offers flexibility in code development for these processors because its compiler, linker, and debugger permit code development in either byte-ordering mode.

Data Alignment Problems, Code

These problems crop up when data elements become misaligned in memory. Put other way, the elements get placed in memory in such a way that they don't match the processor's preferred memory access patterns. Typically, a processor's memory bus is optimized to access memory starting at certain boundaries. (A *boundary* represents a memory location whose starting address is a multiple of a specific data size.) If a processor prefers, say, a 32-bit data alignment, then data items should start at addresses that are multiples of four. Most processors can access data types along their "natural" alignment: that is, as integral multiples of the data element's size. Put another way, they can readily access bytes on 8-bit boundaries, 16-bit values on 16-bit boundaries, and 32-bit values on 32-bit boundaries.

You'll notice in Figure 1 that extra bytes trail the `mess` and `flags` variables. The compiler generates these so-called *padding bytes* to ensure that `flags` and `header` fall on their respective 16- and 32-bit boundaries. If the padding bytes were absent, the data in `flags` and `header` would straddle these boundaries. In a few words, they would become misaligned.

What problems do misaligned data cause? Certain processors don't have the circuitry to access such out-of-place data, and so they issue an exception. Others can deal gracefully with misaligned data, but at price. At a minimum, it takes one cycle to read the data on one side of the boundary, a second cycle to read the remaining data on the other side of the boundary, and a third cycle to reconstruct the bytes into the proper data type. This

assumes that desired variable can be read in one cycle. If the access causes a cache miss, the delay can be much longer.

The capabilities of today's processors show how important the data alignment issue can be. The 68K embedded processors tolerate misaligned data, while the SPARC and MIPS processors generate an exception if the data is not aligned on its natural boundary. The PowerPC prefers a natural data alignment, yet it can handle misaligned data accesses—if it's in the big-endian mode. When it operates in little-endian mode, misaligned data accesses cause an exception. Even these rules aren't absolute. Some PowerPC instructions, typically the floating-point ones, require aligned data, while certain embedded PowerPC processors use an enhanced 603e core that tolerates misaligned data, even in the little-endian mode.

To recap, at worst misaligned data causes an exception, requiring that you dodge the problem by keeping the data aligned. At best, misaligned data triples the time required to access it, so for the sake of performance you want to manage data alignment. Either way, it behooves you carefully consider how a program's data gets placed in memory while doing the code port.

CodeWarrior provides several sophisticated pragmas that let you selectively control the data alignment of structures. For the x86 and MIPS CodeWarrior compilers, you can change the default data alignment on the fly as the program declares data items. Some of the pragmas are summarized in Table 1.

Table 1: CodeWarrior pragmas for managing data alignment.

PRAGMA NAME	PURPOSE	SUPPORTED COMPILERS
align option = alignment type	*Aligns data structures to 2- or 4 byte boundaries, or to natural boundaries.*	*68K, PowerPC*
align_array_members on | off | reset	*Controls data alignment in structs and classes. The data item sizes are specified by the align pragma.*	*68K, PowerPC*
pack (n | push,n | pop)	*Aligns structures to the specified data item size stated by the value in n. Push and pop arguments allow a compiler to adjust data alignment on the fly.*	*x86, MIPS*

GNU C has a language extension, the `__attribute__` keyword, which allows you to assign special characteristics to variables. You'll often see declarations such as `__attribute__((packed))` and `__attribute__((aligned (x)))`, that specifically manage the data alignment of structures. These are used instead of `#pragma align/#pragma pack` because the characteristic can be applied to individual fields in a data structure, like so:

```
typedef struct
{
     char a  __attribute__((packed));
/* "binds" a to next field */
     short x;
}    mess;
```

or

```
typedef struct
{
     char a  __attribute__((aligned (2));
/* gives "a" a 2-byte field */
     short x;
}    mess2;
```

CodeWarrior's pragmas don't permit this level of fine-grained control. To port the above structures, you wrap them with `#pragma options align=pack/reset` pairs and manually add any padding bytes to handle special alignment needs. In CodeWarrior, the declarations of the `mess` and `mess2` structures become:

```
#pragma options align=packed
typedef struct
{
     char a;
     short x;
}    mess;
#pragma options align=reset
```

and

```
#pragma options align=packed
typedef struct
{
     char a;
     char __byte1;       /* padding */
     short x;
}    mess2;
#pragma options align=reset
```

An alternate way to declare the `mess2` structure in CodeWarrior is

```
#pragma options align=2
typedef struct
{
     char a;
     long x;
}    mess2;
#pragma options align=reset
```

Data Alignment Problems, Interfaces

Certain operating systems, such as Solaris, enforce a particular data alignment for the arguments passed to their APIs. If a ported program passes misaligned arguments to a system call, you'll be fortunate if the program crashes immediately. Often, the out of whack system call can fester for some time before the operating system craters some thousands of cycles later (bad, because it's makes debugging difficult). Or, the operating system returns spurious results, causing erratic program behavior (worse), or the program provides inaccurate answers (really, really, bad if it happens to be controlling the tension in a factory's conveyor belt). Look closely at the interface header files for the target operating system to confirm if its system calls require a specific data alignment.

Another problem area for ported programs is the data files made by the original program. If the file contains data structures that the original program wrote from memory directly to disk, watch out for alignment problems when the ported program retrieves the data. This is particularly important if the processor is unforgiving on data alignment, or the retrieved structure will be jammed straight into a system call. To fix this problem, you'll either massage the data into the proper form after it has been read—providing the processor tolerates the original structure's data alignment—or rewrite the program's file I/O functions if it doesn't.

Sizeof() Integer Problems

If the GNU and CodeWarrior compilers disagree on the number of bytes required to represent the `int` data type, all sorts of code glitches can surface. As a contrived example, suppose the GNU compiler used to write the original code implements `int` as a two-byte quantity, perhaps for a 16-bit embedded processor. Further suppose that CodeWarrior implements `int` as a 4-byte quantity. When you recompile the program, the change in integer size skews the layout of all data structures. This repositioning creates data alignment problems, and previously reliable pointers to elements within these structures can wind up pointing—for all practical purposes—nowhere.

The ANSI/ISO standard defines the numeric range of `int` such that it's a 16-bit quantity. However, the standard does allow `int` to span a larger range, and so some compilers and platforms define `int` as a 32-bit quantity. The bottom line is that you must determine the size of `int` before you do a code port. CodeWarrior compilers default to 16 bits for `int`, but they can let you have it both ways. For example, its PowerPC compiler lets you assign `int` to be 2 or 4 bytes in size by changing a setting in a preference panel.

Likewise, don't assume that the value returned by the `sizeof()` operator is assignment compatible with either `int` or `long`. The ANSI/ISO C language standard only requires that `sizeof()` return a *data type* of `size_t`, as defined in the header file `stddef.h`. The standard allows the actual size to be defined by the compiler implementation. Although the data types of `int` and `size_t` are often the same size, sometimes they differ when the code jumps platforms/processors.

The solution to many of these problems is to avoid the use of the generic and potentially ambiguous `int` data type. Replace the existing occurrences of it with size declarations of either `short` (16 bits) or `long` (32 bits) data types. Be aware that the standard also allows `short` and `long` to legally represent larger values. However, it's not as common for a compiler vendor to tinker with these definitions as is the case for `int`. If you suspect problems with these data types, check the numeric ranges for them in `limits.h` file for the GNU compiler.

Implementation Issues Summary

If the GNU code port to CodeWarrior involves moving to a new processor, round up the usual suspects before proceeding: what byte-ordering scheme does the processor use, and what are its data alignment preferences. If the port uses the same platform/processor, you won't see many of the problems described here. However, be alert for possible changes in the interface to the C and system libraries.

Here's a brief checklist to handle porting problems due to implementation changes:

- *If the target processor uses a different byte-ordering scheme, check out all references to variables larger than a* `char`. Variable accesses that rely on pointers, or array references with fixed offsets are suspect. Try to isolate and revise the functions that do bit manipulations. Modify the masks that test or set bits in variables whose data type is larger than `char`.
- *Watch out for misaligned data*. Revise the data alignment of structures by hand or with pragmas when the target processor uses a different data alignment. At the very least, any fixes to the data alignment will improve code performance.
- *Check to see if the interfaces to the target's C libraries or the supported system calls mandate a certain data alignment*.
- *Banish the generic* `int` *declaration from your C language lexicon*. Use `short` and `long` declarations instead.

Language Issues

It might come as a shock to learn that the GNU and CodeWarrior C compilers are based on the same C language standard. This standard was defined by an American National Standards Institute (ANSI) document, *American National Standard X3.159-1989—ANSI C*. The document was later slightly revised by the International Standards Organization (ISO), to become the *ISO/IEC 9899:1990, Programming Languages—C* standards document (henceforth termed "ISO C" for the rest of this paper). The standard specifies the syntax and constraints of the C programming language, and the semantic rules a compiler abides by when it interprets a C program. This raises a valid question: if both compilers started with the same language standard, why does this porting guide exist?

In response to that question, the key word here is "started." Compiler implementations can drift from the standard over time as they support new extensions to the language. Nor will every compiler implement any or all of the updates to the current ISO C standard (there have been several, with the last one in 1995). In addition—for various reasons—compiler vendors may or may not elect to support all of the features present in the existing standard, or in an upcoming one. For example, the CodeWarrior compilers implement certain features found in the draft ISO C9X standard, such as the hyperbolic trigonometric functions. These out-of-step variances among compilers create many of the language-specific problems that plague a code port.

Another source of code incompatibilities are the "loopholes" in the standard. The standard permits certain parts of the language to be implementation-specific. For example, a bitfield variable of the generic data type `int` can be either unsigned or signed; the compiler can implement it either way and not violate the standard. Unfortunately, bitfield operations are a staple of embedded code work, so this one small detail can create headaches for engineers porting embedded programs.

Finally, GNU C offers extensions to the language. Typically, they provide capabilities that are absent from the standard. However, while these extensions are valuable in writing C programs, they aren't very portable.

What follows are brief descriptions of common porting problems based on the GNU compiler's default behavior and its language extensions.

Variations on a Standard

GNU C typically defaults towards a more loose or relaxed interpretation of the ISO C standard. There is a good reason to do so: GNU C must often translate legacy code that predates any standard. CodeWarrior takes the opposite tack by strictly adhering to the standard. However, CodeWarrior has controls that allow you to relax certain syntax/semantic rules so that these same non-standard legacy programs will compile as well.

Because of the GNU compiler's default behavior, it's possible to inadvertently write C code that doesn't quite conform to the standard. Also, recall that some porting problems may arise because of variances in implementing the language standard(s).

Multiple defines

GNU allows you to redefine symbols anywhere in the program. CodeWarrior does not. One solution to this problem is to locate the source line with the multiple define error, and just before it place an `#undef` directive with the offending symbol. This undefines the symbol so that it can be reused.

Function Prototypes

GNU C allows functions that lack prototype declarations to compile. This capability is useful for legacy code where such prototype declarations are absent. CodeWarrior, per the standard, requires the prototypes. However, this requirement can be relaxed by disabling the Require Function Prototypes item in CodeWarrior's C/C++ Language preference panel.

A related issue is that GNU C allows a function prototype to override a subsequent function definition that lacks a prototype. Specifically, this behavior overrides the data types of the function's arguments. CodeWarrior enforces strict type conformance between a function and its prototype, and this checking can't be disabled. In this case, the best solution is to review the suspect functions and write corrected prototype declarations and arguments for them.

Implicit Type Conversions

Consider the following pierce of code:

```
float pi = 3.1459;
float area;
long radius;

/*(radius * radius) = radius squared */
area = 2 * pi * (radius * radius);
```

In this code fragment, before the code performs the calculation, both the digit `2` and the variable `radius` must be converted to the same floating-point data type that `radius` and `pi` uses. As a constant, the digit `2` can be converted at compile time, but `radius`'s conversion requires run-time code.

All compilers, including CodeWarrior, automatically insert the run-time conversion code. The standard requires such implicit type conversions. This feature is fine for general program code, but it can get you into trouble when dealing with pointers. It's possible to mix up the data types for pointer assignments to elements within complicated data

structures. Any automatic conversion in these situations might generate unexpected results.

CodeWarrior will, by default, enforce type checking on pointer assignments and require you to add type casts to such statements. It's strict pointer type checking can be switched off via the Relaxed Pointer Type Rules item in its C/C++ Language preference panel. However, you should double-check the pointer types and the data elements being pointed to, in case such assignments create data alignment problems.

Extending the Language

Like all compiler vendors (including Metrowerks), the GNU Project added extensions to the C language. These extensions provide needed features, such as flexible initialization syntax for arrays and structures, and support for complex numbers. While useful, these extensions can create code porting problems. Some of these extensions will be easy to revise, while others don't have any matching equivalent under the ISO C standard.

Pointer Arithmetic

GNU C lets you mix signed and unsigned pointers, like so:

```
long * sample_Function(long *someAddress)
{
  long *signedPtr;
  signedPtr = someAddress;
  Needs_Unsigned_Pointer(signedPtr);
  return signedPtr;
}
```

where the code passes a `long` pointer into a function requesting an `unsigned long` pointer. In CodeWarrior, you have to explicitly cast `signedPtr` as an `unsigned long *`. The revised code is:

```
long * sample_Function(long *someAddress)
{
  long *signedPtr;
  signedPtr = someAddress;
  Needs_Unsigned_Pointer((unsigned long *)signedPtr);  /*we
cast it*/
  return signedPtr;
}
```

GNU also supports addition and subtraction operations on pointers to `void` and pointers to functions. Such operations aren't allowed in CodeWarrior.

Macro Mayhem

GNU offers language extensions that are useful for writing macros. In GNU C, you can write macros that implement a function, like so:

```
#define INCR(x) ({x++; \
         x;})
```

which can then be used in-line in a program as

```
y = INCR(z);
```

In the above example, GNU lets a compound statement enclosed within parentheses act as an expression—that is, the compound statement produces a value. To backtrack for a moment, a compound statement is a sequence of C statements within curly braces. Note that lone variable in the macro's last statement: it acts as the return value (of the proper data type) for the compound statement. Without it, the compiler would return a value of type `void` for any executable statement that appears in the macro's last line. CodeWarrior doesn't support the use of compound statements this way.

Here's a more concrete application this extension. The following GNU C macro implements a MIPS assembly language function. It sets up an operation for the CPU's custom coprocessor, which then returns a value:

```
#define cop_func( r0 ) __asm__ volatile ( \
   "lwc2 $0, 0( %0 );"                    \
   "lwc2 $1, 4( %0 )"                     \
   :                                      \
   : "r"( r0 ) )
```

As before, the macro statements are enclosed in parentheses so that the caller gets a result. Notice how the final line helps package the return value so that it can be passed out of the macro via the MIPS register `r0`.

You'll often see this language extension used to write macros that splice inline assembly language code into C programs. While CodeWarrior doesn't support this extension, it does have its own assembly language interfaces that enable such macros to be ported. (Note: not all CodeWarrior compilers support the use of inline assembly language in the same way.)

The same macro function, as implemented in CodeWarrior, is

```
#define cop_func( r0 )              \
   __evaluate (__arg0, (long)(r0)); \
   __asm_start();                   \
   __I_lwc2 (0, 0, __arg0);         \
   __I_lwc2 (1, 4, __arg0);         \
   __asm_end();
```

The first statement, `__evaluate`, is itself a macro that defines the function's interfaces, specifying that `ro` is returned as a `long`. The macros `__asm_start()`and `__asm_end()` bracket the actual assembly-language statements, and tweak the environment so that values can be safely passed to/from the assembly code. CodeWarrior's MIPS inline assembler has a good match to GNU's MIPS assembly code, line for line, although the syntax is different. Obviously, such assembly language macros must be heavily reworked to conform to CodeWarrior's style, but at least much of the original code can be salvaged in the effort.

GNU C also allows you to write macros that have a variable number of arguments. For example:

```
#define FORMAT_DATA(format, args...)\
sprintf(buffer, format, ## args...)
```

`Args` contains the arguments passed to the macro, separated by commas. GNU C's preprocessor uses the token ## to discard `format`'s trailing comma if `args` is empty. CodeWarrior C doesn't support this language extension.

Zero-and Variable-length Arrays

The ISO C standard doesn't let you declare arrays of zero length, the logic being that if you're declaring an array, you want to allocate memory for it. GNU C allows you to declare a zero-length array, which is convenient for constructing a pointer to a buffer. As an example, consider

```
struct scanLine {
long sizeX, sizeY;
unsigned char imageData[0];
};
```

Here, the structure acts as a header for a variable-sized element. In ISO C, you must declare `imageData`'s array with a size of 1. CodeWarrior does let this type of declaration stand if the ANSI Strict item in its C/C++ Language preference panel is disabled (the default).

GNU C also permits the declaration of local, dynamically-sized arrays. That is, the value enclosed in the square brackets can be a variable, rather than a constant expression, like so:

```
short munge_Image(sizeX, sizeY, mode)
{
unsigned char imageBuff [sizeX, sizeY];
   ...
/* Do something with imageBuff's contents */
   ...
```

```
return errCode;
}
```

The program retains the memory of the variable-length array `imageBuff` only for the duration of the compound statement (that is, while the thread of execution is in the function block in this example). The usefulness of this feature has spawned a number of language variants, most of them incompatible.

The bad news is that ISO C standard doesn't permit such declarations. The good news is that the C9X standard specifies a uniform method of declaring such variable-length arrays, so that programmer can access this capability in a consistent and portable form. CodeWarrior sides with the current ISO C 9899:1990 standard and doesn't allow this extension. One possible workaround is to determine the largest possible size `imageBuff` can be, and plug that value into the array declaration. Another is to use the `alloca()` function to dynamically allocate memory of the required amount. The `alloca()` function is a language extension that dynamically allocates memory, and supported by many compilers, including CodeWarrior.

Element Initializers

There are two GNU C extensions that manage the initialization of arrays and structures, The first extension supports non-constant initializers for structures. The second extension allows labeled elements in C's initialization expressions. Neither ISO C or CodeWarrior support these extensions.

As an example of non-constant initialization, consider the code sequence

```
typedef struct
{
long a;
long b;
} theStruct;

void the_Funct(long a)
{
theStruct  tempStruct = {a, 4};

...

}
```

Here, the program initializes a part of `tempStruct` using the variable `a`, whose value is known only at run time. Essentially, this code mimics a C++ constructor. ISO C and CodeWarrior require that initializers use constant values, so there's some work to be done to port this type of code.

Now let's consider an example from the second extension. ISO C requires the elements of an array or structure to be initialized in a specific order. The sequence of initialization expressions must match the order of the elements in the array or structure. GNU C, on the other hand, lets you to initialize such elements in any order. You do this by specifying the index of the array element or the structure name, followed by the value. Here's a sample code fragment that sets up an array:

```
float deconvolveVals[10] = {[2] 0.18, [4] 0.25, [5] 0.5,
[6] 0.78, [8] 0.38};
```

The ISO C/CodeWarrior equivalent is:

```
float deconvolveVals[10] = {0.0, 0.0, 0.18, 0.0, 0.25, 0.5,
0.78, 0.0, 0.38, 0.0};
```

For structures, assume a program uses the following data structure:

```
struct {
unsigned char red;
unsigned char green;
unsigned char blue;
} rgbColor;
```

In GNU C, this can be initialized as

```
struct pixel rgbColor = { blue: 255, red: 125, green: 0};
```

For CodeWarrior, you'd write:

```
struct pixel rgbColor = {125, 0, 255};
```

As these examples show, you can easily revise such initialization sequences to conform to the ISO C standard.

Complex Numbers and Double-Length Integers

GNU C has extensions that provide two additional data types: `__complex__` and double-length integers. Complex numbers consist of a real and imaginary component, and are used in many engineering and scientific computations. GNU C allows you declare complex numbers of any data type, but since complex numbers usually involve floating-point calculations, you can expect them to appear declared as `float` or `double`, like so:

```
__complex__ float altCurrent;
__complex__ double fieldStrength;
```

If you're examining the program code to see how much it relies on the `__complex__` data type, be aware that the following operators provide support for complex math calculations:

```
__real__ /* Extracts real component of complex number */
__imaginary__ /* Extracts imaginary component of complex
number */
```

Also look for declarations of `__complex__` constants, such as `altCurrent = 3.0fi` and `fieldStrength = 0.81j`, where the suffixes `i` and `j` denote the real and imaginary components of a complex number, respectively. (The use of the `i` and `j` suffixes to represent a complex number's components is common in electrical engineering and physics work.)

GNU C implements double-length integers, which are twice the size of the default `int` type. For GNU C, these double-length integers are 64 bits in length.

To track down occurrences of double-length integers present in a program, search for `long long` and `unsigned long long` declarations. Like the complex numbers, look for constant declarations with suffixes of `LL` (`signed long long`) and `ULL` (`unsigned long long`). Interestingly, it's game machines such as the MIPS-based Nintendo N-64 and Sony Playstation—not desktop computer applications—that make heavy use of `long long` integers.

CodeWarrior's support for these two GNU extensions is mixed. The MIPS, PowerPC, x86, and 68K versions of the CodeWarrior compiler supports the 64-bit `long long` and `unsigned long long` data types. (Other CodeWarrior embedded compilers do not implement this data type at present.) Complex number calculations aren't implemented in the CodeWarrior C compiler at this time. Support for complex numbers has become part of the draft C9X standard, but that's of no help to you in porting such code at the moment. A more practical solution is to use CodeWarrior's C++ library, which has a standard template that supports this data type. While adding a C++ class to a C program can be a coding headache, such a job is easier than trying to roll your own complex library functions.

Language Issues Summary

These problems fall into the two categories described above: variances in the language and the use of extensions. Here's the checklist for recognizing these problems when migrating the program to CodeWarrior:

- *Write function prototypes for legacy code that lacks them*. Sure, it's more work, but since you're porting the code anyway, now's the time to clean up those interfaces. The prototypes will self-document the function interfaces (since we all comment our code, right?), make you more familiar with the code, and can actually weed out problems where other source files accidentally call the functions using mangled arguments.

- *When dealing with mixed data types, be sure to correctly cast pointers to data structures*. You'd want do this anyway to spot potential data alignment problems that can cripple code performance.
- *Check and clean up any operations that perform calculations on pointers*.
- *Trouble-shoot all macros*. For macros that implement an in-line function, these can be revised. If any of them implement variable-length arguments, the macros will probably have to be replaced with standard code.
- *Watch out for the language extensions*. As this section shows, some of the extensions can be easily revised to be compliant with the standard (and thereby CodeWarrior), while others have no counterpart.

You can disable a certain amount of type-checking that the CodeWarrior compiler performs so that you can quickly compile legacy programs and scout out trouble elsewhere in the code (such as data alignment problem, or a missing library call). If resources permit, however, this would be a good time to overhaul the code so that it conforms to the standard.

Library Issues

Most of the porting problems in this category are due to missing library functions. Differences in the header files that define the interfaces to the library can play a role as well.

To understand the problem involving absent or incompatible library calls, an explanation is in order. The GNU C compiler attempts to support what's known as a *conforming freestanding implementation*. What this mind-numbing term means is that the compiler implements all of the C language features defined by the standard. The C library services are confined to those described by the header files `float.h`, `limits.h`, `stdarg.h`, and `stddef.h`. A program that operates within these boundaries is known as a *strictly conforming program*. Lacking the I/O and utility functions defined in `math.h`, `stdio.h`, and `stdlib.h`, strictly conforming programs won't do much of interest. However, a conforming implementation also allows for extensions that supply the additional library functions. Programs built with these additional functions are known as *conforming programs*, and may—per the definition—not be portable.

Typically, the vendor of the host operating system implements the C library. One problem in this area is that if the library calls don't follow the ISO C standard, then such calls will need revising, no matter which compiler you port the program to.

A related problem is that the GNU C library has been ported to a variety of workstations and is thus compatible with many UNIX-style operating systems. As such, it provides an amalgam of ISO C, BSD, POSIX, and System V service calls. CodeWarrior, on the other hand, started as a development tool for desktop systems. It is also a conforming freestanding implementation. CodeWarrior supplies its own C library, called the Metrowerks Standard Library (MSL), which implements the various I/O services defined

by the standard. The interfaces to these functions comply with the ISO C standard. In recognition that many C programs rely on UNIX calls, MSL also provides a subset of UNIX functions, including some from POSIX. The behavior of these UNIX support functions may vary slightly, depending upon the platform.

MSL is crafted to be platform agnostic in that it provides the same standards-compliant interfaces whether you're porting programs to a Windows desktop computer, a Macintosh, or a Solaris workstation. With the exception of the additional UNIX functions, to obtain access to the host system's APIs you must deliberately include a separate set of header files and libraries.

In summary, both compilers are conforming freestanding implementations. Both GNU C and CodeWarrior supply C libraries that provide needed I/O services, sometimes through use of the host system's APIs. As such, the library interfaces and any conforming program—as allowed by the definition— may not be portable, and therein lies the source of most of the library problems.

Header File Headaches

Many of the facilities the GNU C library provides are compatible with the ISO C standard. However, per the GNU documentation, the library as a whole doesn't attempt to be compliant with the standard. It also offers a superset of functions that provide access to various UNIX system services, and to support its many language extensions. In short, some of the header files declared in a GNU C program may not have a counterpart in the MSL header files, especially if the GNU header file defines a language extension. Whether or not you can work around the problem depends what the missing header file defines. You can substitute some system calls, but language extensions that use special data types such as a complex numbers will be difficult to replace.

Certain GNU C installations provide special header files, known as "fixed header" files. These customized files correct system dependencies present in the operating system's original header files, or they massage certain system calls to be ISO C compliant. However, be aware that some C library calls might still have built-in system dependencies that are reflected in the header files, fixed or not. Such dependencies might be the result of data alignment requirements mandated by the host system. Quite often these dependencies affect the data types of the call's arguments, and can be uncovered when you have CodeWarrior compile the program with Require Function Prototypes enabled.

Missing Library Calls

As the description of the GNU C library indicates, it provides function calls that implement various language extensions—plus a slew of UNIX service calls. Of course, many of these extensions won't be found in the MSL. Handling such extensions requires writing your own support code, or eliminating the program's use of the extension entirely (if possible).

If the GNU C program makes heavy use of UNIX services, you'll want to consult Table 2 to determine how many of these calls are present in the MSL. For comparison, the table includes all of those calls supported by ISO C. The calls are arranged alphabetically so that you can quickly determine whether or not the MSL implements the function.

For the UNIX service calls, the MSL might provide a counterpart. The UNIX and POSIX calls supported by MSL can be found in the header files `fnctl.platform.h`, `stat.platform.h`, `unistd.platform.h`, `unix.h`, `utime.platform.h`, and `utsname.h`, where `platform` is the name of the target system, such as `mac` or `win32`. You can also consult the Metrowerks' *MSL C Library Reference* manual. If the function call in question is absent, it might be reworked to use a UNIX call that the MSL supports. The number of unsupported calls should help you determine the scale of the porting job. Due to Table 2's length, it's presented at the end of this paper.

Library Issues Summary

Here's the checklist for handling porting problems related to libraries:

- *Check for missing header files*. This will give you a clear picture as to what services the GNU C program is using. Some of the missing services can be substituted, while others will require substantial work.
- *Have CodeWarrior use function prototypes to uncover data type problems for those service calls that MSL does support*. This can uncover potential problems where a service call uses a data type different from the MSL service call. Use Table 2 to locate the appropriate MSL header file for the call and check the function's prototype for discrepancies. This can also uncover possible data alignment problems.
- *Use Table 2 to determine what service calls you'll have to revise to work with the facilities provided by the MSL*. For some of the missing service functions, you might use the APIs provided by the host operating system to implement the facility. Because of the many variations of interfaces (just in the GNU C library alone), combined with those presented by the host operating system, it's difficult to provide specific guidelines on how to make the substitution.

Any Safe Port

As can be seen, while the ISO C standard goes a long way towards promoting portable code, there are enough loopholes in it to allow porting problems to exist. Variations in how the libraries support certain facilities can also create problems. Some of these incompatibilities comprise the use of system calls that can be worked around. Other are extensions that provide useful features not present in the standard. All of these factors introduce dependencies in the program code that can make it hard to port. Having said that, it's worth pointing out that following the standard can ease the job of porting code. GNU C programs that conform to the C9X standard and use no language extensions have been "ported" to CodeWarrior by just recompiling the code.

The best way to approach a porting job is to first determine how much the source code is out of compliance with the ISO C standard. It's worth repeating that the code's lack of compliance is not the fault of any compiler, but because the program probably consists of legacy code written before the standard existed. Both GNU and CodeWarrior offer relaxed checking so that such programs can compile.

To make this determination, use the GNU compiler's `-ansi` option, or enable ANSI strict in CodeWarrior's C/C++ Language preference panel. The warning and error messages should point you to problem spots in the code. This information, along with any error reports about the header files, should also indicate whether language extensions were used.

Use the information in this document to help you determine what substitutions can be made, and gauge the difficulty in revising certain extensions. For example, certain initialization features in GNU C can be easily rewritten, macros using variable arguments will be harder, and it may take a major coding effort to support complex numbers.

Finally, while porting the code, try to conform to the ISO C standard. While it's hard to plan ahead while migrating code to a new compiler, by sticking to the standard you make it possible for the program to be ported to new hardware in the future. If the Y2K bug has taught us anything, it's that reliable, mission-critical code can have a lifetime measured in the decades. CodeWarrior's strict following of the standard, plus its wide support of different processors and platforms might help your program survive the ravages of time.

Table 2: Summary of C functions supported by the MSL.

FUNCTION	ISO C STANDARD	METROWERKS STANDARD LIBRARY	MSL HEADER FILE
`abort()`	✓	✓	*<cstdlib>*
`abs()`	✓	✓	*<cstdlib>*
`acos()`	✓	✓	*<cmath>*
`acosh()`[9]		✓	*<cmath>*
`alloca()`		✓	*alloca.h*
`asctime()`	✓	✓	*<ctime>*
`asin()`	✓	✓	*<cmath>*
`asinh()`[9]		✓	*<cmath>*
`assert()`	✓	✓	*cassert.h*
`atan2()`	✓	✓	*<cmath>*
`atan()`	✓	✓	*<cmath>*
`atanh()`[9]		✓	*<cmath>*
`atexit()`	✓	✓	*<cstdlib>*
`atof()`	✓	✓	*<cstdlib>*
`atoi()`	✓	✓	*<cstdlib>*
`atol()`	✓	✓	*<cstdlib>*
`bsearch()`	✓	✓	*<cstdlib>*
`btowc()`[1]	✓	*planned*	
`calloc()`	✓	✓	*<cstdlib>*
`ceil()`	✓	✓	*<cmath>*
`chdir()`		✓	*unistd.platform.h*
`clearerr()`	✓	✓	*<cstdio>*
`clock()`	✓	✓	*<ctime>*
`close()`		✓	*unistd.platform.h*
`copysign()9`		✓	*<cmath>*
`cos()`	✓	✓	*<cmath>*
`cosh()`	✓	✓	*<cmath>*
`creat()`		✓	*fcntl.platform.h*
`ctime()`	✓	✓	*<ctime>*
`cuserid()`		✓	*unistd.platform.h*
`difftime()`	✓	✓	*<ctime>*
`div()`	✓	✓	*<cstdlib>*

1: Function defined in the ISO/IEC 9899:1990/Amendment 1:1995(E).

9: Function is part of the draft ISO/IEC C9X standard.

Table 2: Summary of C functions supported by the MSL (continued).

FUNCTION	ISO C STANDARD	METROWERKS STANDARD LIBRARY	MSL HEADER FILE
`erf()`[9]		✓	*<cmath>*
`erfc()`[9]		✓	*<cmath>*
`execl()`		✓	*unistd.platform.h*
`execle()`		✓	*unistd.platform.h*
`execlp()`		✓	*unistd.platform.h*
`execv()`		✓	*unistd.platform.h*
`execve()`		✓	*unistd.platform.h*
`execvp()`		✓	*unistd.platform.h*
`exit()`	✓	✓	*<cstdlib>*
`exp2()`[9]		✓	*<cmath>*
`exp()`	✓	✓	*<cmath>*
`expm1()`[9]		✓	*<cmath>*
`fabs()`	✓	✓	*<cmath>*
`fclose()`	✓	✓	*<cstdio>*
`fcntl()`		✓	*fcntl.platform.h*
`fdim()`[9]		✓	*<cmath>*
`fdopen()`		✓	*<cstdio>*
`feof()`	✓	✓	*<cstdio>*
`ferror()`	✓	✓	*<cstdio>*
`fflush()`	✓	✓	*<cstdio>*
`fgetc()`	✓	✓	*<cstdio>*
`fgetpos()`	✓	✓	*<cstdio>*
`fgets()`	✓	✓	*<cstdio>*
`fgetwc()`[1]	✓	*planned*	
`fgetws()`[1]	✓	*planned*	
`fileno()`		✓	*<cstdio>*
`floor()`	✓	✓	*<cmath>*
`fmax()`[9]		✓	*<cmath>*
`fmin()`[9]		✓	*<cmath>*
`fmod()`	✓	✓	*<cmath>*
`fopen()`	✓	✓	*<cstdio>*
`fpclassify()`[9]		✓	*<cmath>*

1: Function defined in the ISO/IEC 9899:1990/Amendment 1:1995(E).

9: Function is part of the draft ISO/IEC C9X standard.

Table 2: Summary of C functions supported by the MSL (continued).

FUNCTION	ISO C STANDARD	METROWERKS STANDARD LIBRARY	MSL HEADER FILE
`fprintf()`	✓	✓	*<cstdio>*
`fputc()`	✓	✓	*<cstdio>*
`fputs()`	✓	✓	*<cstdio>*
`fputwc()`[1]	✓	*planned*	
`fputws()`[1]	✓	*planned*	
`fread()`	✓	✓	*<cstdio>*
`free()`	✓	✓	*<cstdlib>*
`freopen()`	✓	✓	*<cstdio>*
`frexp()`	✓	✓	*<cmath>*
`fscanf()`	✓	✓	*<cstdio>*
`fseek()`	✓	✓	*<cstdio>*
`fsetpos()`	✓	✓	*<cstdio>*
`fstat()`		✓	*stat.platform.h*
`ftell()`	✓	✓	*<cstdio>*
`fwide()`[1]	✓	✓	*<cstdio>*
`fwprintf()`[1]	✓	*planned*	
`fwrite()`	✓	✓	*<cstdio>*
`fwscanf()`[1]	✓	*planned*	
`gamma()`[9]		✓	*<cmath>*
`getc()`	✓	✓	*<cstdio>*
`getch()`		✓	*console.h*
`getchar()`	✓	✓	*<cstdio>*
`getcwd()`		✓	*unistd.platform.h*
`getegid()`		✓	*unistd.platform.h*
`getenv()`	✓	✓	*<cstdlib>*
`geteuid()`		✓	*unistd.platform.h*
`getgid()`		✓	*unistd.platform.h*
`getlogin()`		✓	*unistd.platform.h*
`getpgrp()`		✓	*unistd.platform.h*
`getpid()`		✓	*unistd.platform.h*
`getppid()`		✓	*unistd.platform.h*
`gets()`	✓	✓	*<cstdio>*

1: Function defined in the ISO/IEC 9899:1990/Amendment 1:1995(E).

9: Function is part of the draft ISO/IEC C9X standard.

Table 2: Summary of C functions supported by the MSL (continued).

FUNCTION	ISO C STANDARD	METROWERKS STANDARD LIBRARY	MSL HEADER FILE
`getuid()`		✓	*unistd.platform.h*
`getwc()`[1]	✓	*planned*	
`getwchar()`[1]	✓	*planned*	
`gmtime()`	✓	✓	*<ctime>*
`hypot()`[9]		✓	*<cmath>*
`isalnum()`	✓	✓	*<cctype>*
`isalpha()`	✓	✓	*<cctype>*
`isatty()`		✓	*unistd.platform.h*
`iscntrl()`	✓	✓	*<cctype>*
`isdigit()`	✓	✓	*<cctype>*
`isfinite()`[9]		✓	*<cmath>*
`isgraph()`	✓	✓	*<cctype>*
`isgreater()`[9]		✓	*<cmath>*
`isgreaterequal()`[9]		✓	*<cmath>*
`isless()`[9]		✓	*<cmath>*
`islessequal()`[9]		✓	*<cmath>*
`islessgreater()`[9]		✓	*<cmath>*
`islower()`	✓	✓	*<cctype>*
`isnan()`[9]		✓	*<cmath>*
`isnormal()`[9]		✓	*<cmath>*
`isprint()`	✓	✓	*<cctype>*
`ispunct()`	✓	✓	*<cctype>*
`isspace()`	✓	✓	*<cctype>*
`isunordered()`[9]		✓	*<cmath>*
`isupper()`	✓	✓	*<cctype>*
`iswalnum()`[1]	✓	✓	*<cwctype>*
`iswalpha()`[1]	✓	✓	*<cwctype>*
`iswblank()`[9]		✓	*<cctype>*
`iswcntrl()`[1]	✓	✓	*<cwctype>*
`iswctype()`[1]	✓	✓	*<cwctype>*
`iswdigit()`[1]	✓	✓	*<cwctype>*
`iswgraph()`[1]	✓	✓	*<cwctype>*

1: Function defined in the ISO/IEC 9899:1990/Amendment 1:1995(E).

9: Function is part of the draft ISO/IEC C9X standard.

Table 2: Summary of C functions supported by the MSL (continued).

FUNCTION	ISO C STANDARD	METROWERKS STANDARD LIBRARY	MSL HEADER FILE
`iswlower()`[1]	✓	✓	*<cwctype>*
`iswprint()`[1]	✓	✓	*<cwctype>*
`iswpunct()`[1]	✓	✓	*<cwctype>*
`iswspace()`[1]	✓	✓	*<cwctype>*
`iswupper()`[1]	✓	✓	*<cwctype>*
`iswxdigit()`[1]	✓	✓	*<cwctype>*
`isxdigit()`	✓	✓	*<cctype>*
`labs()`	✓	✓	*<cstdlib>*
`llabs()`		✓	*<cstdlib>*
`ldexp()`	✓	✓	*<cmath>*
`ldiv()`	✓	✓	*<cstdlib>*
`lgamma()`[9]		✓	*<cmath>*
`lldiv()`		✓	*<cstdlib>*
`localeconv()`	✓	✓	*<clocale>*
`localtime()`	✓	✓	*<ctime>*
`log10()`	✓	✓	*<cmath>*
`log1p()`[9]		✓	*<cmath>*
`log2()`[9]		✓	*<cmath>*
`log()`	✓	✓	*<cmath>*
`logb()`[9]		✓	*<cmath>*
`longjmp()`	✓	✓	*<csetjmp>*
`lseek()`		✓	*unistd.platform.h*
`malloc()`	✓	✓	*<cstdlib>*
`mblen()`	✓	✓	*<cstdlib>*
`mbrlen()`[1]	✓	*planned*	
`mbrtowc()`[1]	✓	*planned*	
`mbsinit()`[1]	✓	*planned*	
`mbsrtowcs()`[1]	✓	*planned*	
`mbstowcs()`	✓	✓	*<cstdlib>*
`mbtowc()`	✓	✓	*<cstdlib>*
`memchr()`	✓	✓	*<cstring>*
`memcmp()`	✓	✓	*<cstring>*

1: Function defined in the ISO/IEC 9899:1990/Amendment 1:1995(E).

9: Function is part of the draft ISO/IEC C9X standard.

Table 2: Summary of C functions supported by the MSL (continued).

FUNCTION	ISO C STANDARD	METROWERKS STANDARD LIBRARY	MSL HEADER FILE
`memcpy()`	✓	✓	*<cstring>*
`memmove()`	✓	✓	*<cstring>*
`memset()`	✓	✓	*<cstring>*
`mkdir()`		✓	*stat.platform.h*
`mktime()`	✓	✓	*<ctime>*
`modf()`	✓	✓	*<cmath>*
`nan()`[9]		✓	*<cmath>*
`nearbyint()`[9]		✓	*<cmath>*
`nextafter()`[9]		✓	*<cmath>*
`open()`		✓	*fcntl.platform.h*
`pclose()`		✓	*<cstdio>*
`perror()`	✓	✓	*<cstdio>*
`popen()`		✓	*<cstdio>*
`pow()`	✓	✓	*<cmath>*
`printf()`	✓	✓	*<cstdio>*
`putc()`	✓	✓	*<cstdio>*
`putchar()`	✓	✓	*<cstdio>*
`puts()`	✓	✓	*<cstdio>*
`putwc()`[1]	✓	*planned*	
`putwchar()`[1]	✓	*planned*	
`qsort()`	✓	✓	*<cstdlib>*
`raise()`	✓	✓	*<csignal>*
`rand()`	✓	✓	*<cstdlib>*
`read()`		✓	*unistd.platform.h*
`realloc()`	✓	✓	*<cstdlib>*
`remainder()`[9]		✓	*<cmath>*
`remove()`	✓	✓	*<cstdio>*
`rename()`	✓	✓	*<cstdio>*
`remquo()`[9]		✓	*<cmath>*
`rewind()`	✓	✓	*<cstdio>*
`rint()`[9]		✓	*<cmath>*
`rmdir()`		✓	*unistd.platform.h*

1: Function defined in the ISO/IEC 9899:1990/Amendment 1:1995(E).

9: Function is part of the draft ISO/IEC C9X standard.

Table 2: Summary of C functions supported by the MSL (continued).

FUNCTION	ISO C STANDARD	METROWERKS STANDARD LIBRARY	MSL HEADER FILE
`round()`[9]		✓	*<cmath>*
`scalb()`[9]		✓	*<cmath>*
`scanf()`	✓	✓	*<cstdio>*
`setbuf()`	✓	✓	*<cstdio>*
`setjmp()`	✓	✓	*<csetjmp>*
`setlocale()`	✓	✓	*<clocale>*
`setvbuf()`	✓	✓	*<cstdio>*
`signal()`	✓	✓	*<csignal>*
`signbit()`[9]		✓	*<cmath>*
`sin()`	✓	✓	*<cmath>*
`sinh()`	✓	✓	*<cmath>*
`sleep()`		✓	*unistd.platform.h*
`snprintf()`[9]		✓	*<cstdio>*
`sprintf()`	✓	✓	*<cstdio>*
`sqrt()`	✓	✓	*<cmath>*
`srand()`	✓	✓	*<cstdlib>*
`sscanf()`	✓	✓	*<cstdio>*
`stat()`		✓	*stat.platform.h*
`strcat()`	✓	✓	*<cstring>*
`strchr()`	✓	✓	*<cstring>*
`strcmp()`	✓	✓	*<cstring>*
`strcoll()`	✓	✓	*<cstring>*
`strcpy()`	✓	✓	*<cstring>*
`strcspn()`	✓	✓	*<cstring>*
`strerror()`	✓	✓	*<cstring>*
`strlen()`	✓	✓	*<cstring>*
`strncat()`	✓	✓	*<cstring>*
`strncmp()`	✓	✓	*<cstring>*
`strncpy()`	✓	✓	*<cstring>*
`strpbrk()`	✓	✓	*<cstring>*
`strrchr()`	✓	✓	*<cstring>*
`strspn()`	✓	✓	*<cstring>*

1: Function defined in the ISO/IEC 9899:1990/Amendment 1:1995(E).

9: Function is part of the draft ISO/IEC C9X standard.

Table 2: Summary of C functions supported by the MSL (continued).

FUNCTION	ISO C STANDARD	METROWERKS STANDARD LIBRARY	MSL HEADER FILE
`strstr()`	✓	✓	*<cstring>*
`strtod()`	✓	✓	*<cstdlib>*
`strtok()`	✓	✓	*<cstring>*
`strtol()`	✓	✓	*<cstdlib>*
`strtoul()`	✓	✓	*<cstdlib>*
`strxfrm()`	✓	✓	*<cstring>*
`swprintf()`[1]	✓	*planned*	
`swscanf()`[1]	✓	*planned*	
`system()`	✓	✓	*<cstdlib>*
`tan()`	✓	✓	*<cmath>*
`tanh()`	✓	✓	*<cmath>*
`time()`	✓	✓	*<ctime>*
`tmpfile()`	✓	✓	*<cstdio>*
`tmpnam()`	✓	✓	*<cstdio>*
`tolower()`	✓	✓	*<cctype>*
`toupper()`	✓	✓	*<cctype>*
`towctrans()`[1]	✓	*planned*	
`towlower()`[1]	✓	✓	*<cwctype>*
`towupper()`[1]	✓	✓	*<cwctype>*
`trunc()`[9]		✓	*<cmath>*
`ttyname()`		✓	*console.h*
`uname()`		✓	*utsname.h*
`ungetc()`	✓	✓	*<cstdio>*
`ungetwc()`[1]	✓	*planned*	
`unlink()`		✓	*unistd.platform.h*
`utime()`		✓	*utime.platform.h*
`utimes()`		✓	*utime.platform.h*
`vfprintf()`		✓	*<cstdio>*
`vfwprintf()`[1]	✓	*planned*	
`vprintf()`	✓	✓	*<cstdio>*
`vsnprintf()`		✓	*<cstdio>*
`vsprintf()`	✓	✓	*<cstdio>*

1: Function defined in the ISO/IEC 9899:1990/Amendment 1:1995(E).

9: Function is part of the draft ISO/IEC C9X standard.

Table 2: Summary of C functions supported by the MSL (continued).

FUNCTION	ISO C STANDARD	METROWERKS STANDARD LIBRARY	MSL HEADER FILE
`vswprintf()`[1]	✓	*planned*	
`vwprintf()`[1]	✓	*planned*	
`wcrtomb()`[1]	✓	*planned*	
`wcscat()`[1]	✓	*planned*	
`wcschr()`[1]	✓	*planned*	
`wcscmp()`[1]	✓	*planned*	
`wcscoll()`[1]	✓	*planned*	
`wcscpy()`[1]	✓	*planned*	
`wcscspn()`[1]	✓	*planned*	
`wcsftime()`[1]	✓	*planned*	
`wcslen()`[1]	✓	*planned*	
`wcsncat()`[1]	✓	*planned*	
`wcsncmp()`[1]	✓	*planned*	
`wcsncpy()`[1]	✓	*planned*	
`wcspbrk()`[1]	✓	*planned*	
`wcsrchr()`[1]	✓	*planned*	
`wcsrtombs()`[1]	✓	*planned*	
`wcsspn()`[1]	✓	*planned*	
`wcsstr()`[1]	✓	*planned*	
`wcstod()`[1]	✓	*planned*	
`wcstok()`[1]	✓	*planned*	
`wcstol()`[1]	✓	*planned*	
`wcstombs()`	✓	✓	*<cstdlib>*
`wcstoul()`[1]	✓	*planned*	
`wcsxfrm()`[1]	✓	*planned*	
`wctob()`[1]	✓	✓	*<cwctype>*
`wctomb()`	✓	✓	*<cstdlib>*
`wctrans()`[1]	✓	*planned*	
`wctype()`[1]	✓	✓	*<cwctype>*
`wmemchr()`[1]	✓	*planned*	
`wmemcmp()`[1]	✓	*planned*	
`wmemcpy()`[1]	✓	*planned*	

1: Function defined in the ISO/IEC 9899:1990/Amendment 1:1995(E).

9: Function is part of the draft ISO/IEC C9X standard.

Table 2: Summary of C functions supported by the MSL (continued).

FUNCTION	ISO C STANDARD	METROWERKS STANDARD LIBRARY	MSL HEADER FILE
`wmemmove()`[1]	✓	*planned*	
`wmemset()`[1]	✓	*planned*	
`wprintf()`[1]	✓	*planned*	
`write()`		✓	*unistd.platform.h*
`wscanf()`[1]	✓	*planned*	

1: Function defined in the ISO/IEC 9899:1990/Amendment 1:1995(E).

9: Function is part of the draft ISO/IEC C9X standard.