C and C++ 68000 Family Release Notes

Introduction

These release notes describe Releases 8.3, 8.4, 8.5, 9.0, 9.01, 9.02 and 9.03 of the InterTools 68000 Family Compiler/Assembler Toolkit. The notes are divided into the following sections:

Compiler Back End - Fixed in Release 8.4
Compiler Back End - Fixed in Release 8.3
Assembler - Fixed in Release 8.3
Linking Locator - Fixed in Release 9.03
Linking Locator - Fixed in Release 9.02
Linking Locator - Fixed in Release 9.0
Linking Locator - Fixed in Release 8.3
C Run-time Library - Fixed in Release 9.03
C Run-time Library - Fixed in Release 9.01
C Run-time Library - Fixed in Release 9.0
C Run-time Library - Fixed in Release 8.3
Formatter - Fixed in Release 9.02
Formatter - Fixed in Release 9.0
Formatter - Fixed in Release 8.4
Formatter - Fixed in Release 8.3

Important Changes

Changes in Release 9.03

****

C++ and C compilers now support the following new command-line options to specify the underlying type of an enum: --enum1u, --enum1s, --enum2s, --enum4s. These options specify unsigned char, signed char, signed 16-bit integer (default), and signed 32-bit integer, respectively.

Changes in Release 9.01

****

Release 9.01 of the InterTools Compiler package incorporates some important new features supporting C++. The C++ compiler now supports all existing InterTools extensions:

```
#pragma separate
#pragma sep_on
#pragma sep_off
```

_ASMLINE
_ASM
_CASM
_IH
_SWI
SPL
GPL
TRAP

****

Changes in Release 9.0

****

The InterTools 68K Compiler V9.0 includes the following enhancements:
· support for C/C++ and Embedded C++
· integration with EDE V2.0
· bug fixes

The integration with EDE (Embedded Development Environment) V2.0 provides you with direct and easy access to all the tools you need for your application development via a single user interface.

The InterTools C++ compiler closely conforms to the evolving ANSI X3J16 C++ standard. Templates, dynamic casts, run-time type identification, and exception handling are supported. The 68K CrossView Pro Debugger V3.0 supports C++ debugging.

Embedded C++ is a subset of C++ (included with the C++ InterTools option) that offers upward compatibility with the version of C++, retains the major advantages of C++ yet fulfills the particular requirements of embedded systems designs. The InterTools C++ capability is available as a separate option.

****

Changes in Release 8.5

****

For the PC, two separate products are available for 16-bit and 32-bit environments. The 32-bit variant is a new addition to the set of InterTools products. It can be used under both Windows95 and Windows NT.

****

The Motorola software floating-point package FPSP has been added to the 68040 and 68060 run-time libraries. The sources of this run-time library package are available for downloading from the Motorola web site. The location is:

http://www.mot.com/SPS/General/mp.html

Then, make the following selections:

M680X0 Family of 32-bit Microprocessors
680X0 Family Members
68040 (or 68060)
Engineer's Tool Box
M68040FPSP Code (or M68060FPSP Code)

****

A new feature has been added to the IEEE695 formatter. This option makes it possible to add line information into IEEE695 records for assembly code processed by the compiler.

****

Changes in Release 8.4

****

The installation program shipped with release 8.4 for the IBM-PC is now a Microsoft Windows application. Users upgrading from previous versions of InterTools should note that the default installation directories selected for the InterTools libraries is more streamlined than in earlier compiler releases. The new directory structure eliminates the duplication of libraries when

you install compiler support for multiple processors of the same family.

Since the directories and their default names are somewhat different from earlier versions of the compiler, batch and make files created for use with earlier versions of the compiler will not be compatible with the default installation for Release 8.4.

You can override the default directory structure and re-create the directory structure of the earlier releases.

If you choose not to override the default you should modify any batch files or make files created to work with earlier releases.

Users of the Precise/MQX kernel should note in particular that batch files and make files installed with the MQX kernel (versions 2.30 and earlier) will not work properly and need to be modified to build an MQX library and MQX demo programs.

For example:

> If you install version 8.4 of the compiler to support a Motorola 68332 and you are using Precise/MQX version 2.30 for the same processor, you will have to modify the go.bat files installed with MQX to successfully rebuild the kernel.
>
> The following line in the 68332 section of the file, go.bat:
>
> set i2include=%proc_root%;%itool_root%\rtlibs\lib332\inc
>
> would need to be changed as follows:
>
> set i2include=%proc_root%;itool_root%\rtlibs\lib020s\inc
>
> This change would be required in the go.bat file found in each of the following 68332 MQX directories:
>
> > %MQX_ROOT%\kernel\base\generic
> > %MQX_ROOT%\kernel\base\cpu32_sr
> > %MQX_ROOT%\kernel\mot_evk2

%MQX_ROOT%\kernel\format.io

****

In previous releases, the usual argument conversions (promoting char and short to int, and float to double) were applied to parameters to _ASM macros. Thus, an _ASM macro declared to receive a short would actually receive an int, requiring that the compiler generate additional code to coerce the short into an int. As this defeats the purpose of passing arguments "where they are", the types specified in the _ASM macro declaration now overrule the usual argument conversions.

****

The 8.3.1 compiler added support for automatic inline procedure expansion via the -ai command line switch, but code structured for use with this switch could not be effectively debugged (see Changes in Release 8.3 subtopic for more details). That restriction has been removed in the 8.4 release.

****

The formatter now supports the P+E map file format. P+E map files can be generated via the "-d pe" command line switch, and will have a default filename extension of ".map".

****

The MC68060 target is now supported. This means there is a new compiler (c68060), a new assembler (asm68060), and new run-time libraries to support the hardware floating-point features of this target.

****

Changes in Release 8.3

****

The 8.3.1 compiler supports a limited form of automatic inline

procedure expansion. This optimization is requested by the -ai (automatic inline) command line switch.

A procedure is said to be "expanded inline" if a copy of the body of the subroutine is inserted in place of the usual call operation. Generally speaking, inline procedure expansion represents a trade-off of code space for execution speed. You save the execution time spent in a call and return, but the compiler must generate a whole new copy of the called subroutine body at every inline call.

In the 8.3.1 release, inline expansion cannot be performed if the -d (generate debugging symbols) switch is present. In addition, there are rather severe limits on the kinds of routines which can be expanded inline. Inline expansion is limited to two cases:

a) Static subroutines which return an integral value and which contain no flow-of-control statements, i.e., no loops or "if" statements. (Conditional "?:" expressions are OK.)

b) Static subroutines of type void, i.e., which return no value.

These restrictions will be relaxed in future releases. There is also a limit on the overall size for an inline routine, a limit of 6 arguments in an inline routine.

Here is what the compiler does when the -ai command line switch is present. When the compiler processes a static subroutine, it builds up an internal copy of the body of the subroutine. If the routine turns out to be unsuitable for inline expansion, then the body is emitted as usual. Otherwise no code is emitted and the body is retained for future use. When the compiler processes a procedure call to a routine for which it has kept a copy, then it duplicates the body in place of the call.

At the end of the compilation unit, the compiler checks to see if there is any need to emit an out-of-line body. There are two reasons this might be necessary. First, there may have been calls that were not expanded inline because they appeared before the definitions of the routine. Second, some use might have been made of the address of the function. For example, the address may have been assigned to a pointer-to-function. If neither of these conditions apply, then the saved body is discarded.

To take advantage of inline procedure expansion across compilation units, you may be tempted to place the procedures which you wish to expand inline in a set of ".h" include files, together with their bodies. This allows you to have these procedures expanded inline in many different compilation units.

We must warn you that code structured in this manner cannot be effectively debugged, even if you turn off the -ai switch, which allows you to compile with symbols. (It will compile and run without errors.) The problem is that most debug symbol table formats, including the .xdb file used by xdb and PassKey, cannot describe programs whose code comes from more than one source file. In fact, the formatter treats the first file with line number marks in it as the "primary" source file, and marks from other files are discarded. These restrictions will be removed in later releases of the compiler and debugger, but until then we cannot recommend the methodology of putting source in include files.

****

The meaning and usage of the -ss and -sc switches has changed. In previous releases, the use of the -ss and -sc switches also implied the use of -sd, that is, they made all variables separate. Now these switches define a default segment or class name for separate variables which are not assigned an explicit segment or class. They do not cause variables to become separate. Note that this change affects the library building procedure described on pages 7-23 and 7-25 of the User's Manual. See the Compiler section of the User's Manual for more details.

****

The MC68360 target is now supported. This means there is a new compiler (c68360), a new assembler (asm68360), and new run-time libraries.

****

A new compiler switch, -ih, has been added to direct the compiler to assume that routines called by interrupt handlers do not use floating point arithmetic.

An interrupt routine that uses floating point registers must preserve the state of the floating point unit. This requires several instructions, in the entry/exit sequence starting with an FSAVE. Normally any interrupt routine which makes a subroutine call must do the same. This is necessary because the compiler fears that the called routine may do floating point arithmetic. However, if you know that the routines called by your interrupt handlers do not do any floating point arithmetic, then these saves are unnecessary. They can also be quite slow.

By supplying the -ih switch, you can tell the compiler not to worry about the routines called from your interrupt handlers. Of course, if the interrupt handler itself uses floating point registers, then they will be saved on entry nevertheless.

If the -h switch (use 68881/68882 hardware floating point instructions) is not used, the -ih switch has no effect.

****

A new form of in-line assembly is now provided by the _ASMLINE built-in function. By coding

```
_ASMLINE("string");
```

you cause the compiler to emit the indicated line directly into the generated assembly language (the compiler will append a newline character). This construct may appear within or between procedures. Ordinary C escape processing is NOT performed on the assembly language string, so you can't embed newlines using "\n". Also, remember that in assembly language only labels can begin in column 1, so you probably will want to put a leading blank in your string. The resulting code is otherwise ignored by the compiler.

Many other compilers support this feature using "asm" in place of "_ASMLINE". The InterTools compiler uses _ASMLINE because the ANSI C standard says that C programs should to be able to use "asm" as an identifier. If you prefer "asm", either add "#define asm _ASMLINE" to your program or use the equivalent command line switch, "-P asm=_ASMLINE".

This feature is very simple and straightforward, but it has several weaknesses. First, it cannot receive arguments or return results to the surrounding C code. Second, because it is completely ignored by the optimizer, it must not cause side-effects (such as modifying registers and non-volatile global variables) which could invalidate the results of optimization. In cases where more flexibility is needed, a pre-defined in-line assembly insertion must be used.

****

A new and separate formatter, called form695, has been added to the Toolkit to support the IEEE-695 object module format. See the Formatter chapter of the InterTools User's Manual for more information.

****

New Features

Linking Locator - New in Release 9.0

****

The linker (llink) now supports the -opfile <file> command-line option, where <file> contains an arbitrary number of filenames and linker options.

The C++ linker (ldriver) also supports the -opfile command-line option. As a consequence of this, support for the -i and -il linker switches was not implemented.

****

C Run-time Library - New in Release 8.3

****

Support for C or assembly language access to the on-board peripherals of the 68332, 68340, and 68360 has been added to the run-time library.

See the Support for the On-Board Peripherals of the 68332, 68340, and 68360 application note in the InterTools User's Manual for more information.

****

Formatter - New in Release 9.0

****

The formatter (both form and form695) now supports the -opfile <file> command-line option, where <file> contains an arbitrary number of filenames and linker options.

****

Formatter - New in Release 8.5

****

Now form695 allows the generation of additional assembly debug information that is usable with SDS debugger products. This capability can be accessed via the -d switch as described below.

The IEEE695 specification provides records for line number symbol information at the C level. The standard does not support these records at the assembly language level, however.

The modification generates a dummy high level function block that will give SDS debuggers access to line information for assembly routines. This modification only works for SDS debuggers and produces a non-standard IEEE695 output file.

The following replaces the documentation for the -d option for form695:

-d [abs | asm | asmabs]
form695 only. The -d switch controls the generation of symbolic records. Symbolic records, if generated, are placed into the formatted output file. The debug part of the IEEE-695 output file will be omitted unless

this switch is present. If "abs" is used, all global variables that are group data relative (not separate data) are given absolute addresses in the debug information part of the output file. This may be used to examine global variables even when the static base register has not been initialized. If "asm" is used, assembly routines will generate a high level language block with debug line information. This will provide debug line information usable by SDS debuggers. Using "asmabs" is equivalent to using both "abs" and "asm" as described above.

****

Known Problems

Compiler Front End

****

A spurious warning message was sometimes generated for the conversion of an enumeration type to an integer. [PTM 5681]

****

Not all ANSI error checks for functions returning structures are properly enforced. ANSI C says that a structure returned from a function is not an lvalue. This means that it should be illegal to take its address. The compiler permits this, but the results are unpredictable. [PTM 5351]

****

Several error conditions in source programs are not properly flagged by the front end. Usually these result in internal errors in the back end or optimizer. For example, the `&' (logical and) operator cannot be applied to floating point values. An attempt to do so should result in a front end diagnostic, not a "syntax error" in the back end.

****

Under the -E (print pre-processor output) switch, wide character constants (L'a') do not print properly.

****

Under the -mp (make prototypes) switch, prototypes are generated for static functions. They should be omitted.

****

Linking Locator

When building a project with C++, you may encounter spurious warnings from the linker which can be safely ignored. These warnings will involve unresolved references to the name of the ROM processing segment (if doing ROM processing). These warnings occur because the C++ linker driver typically invokes llink multiple times, so certain symbols are not resolved until the final link.

****

Defects Removed

Compiler Driver - Fixed in Release 9.03

****

A bug which had made it impossible to use a C++ compiler driver (e.g., cp68332) to compile a C program in a subdirectory (e.g., subdir/hello.c) has been fixed.

****

Compiler Driver - Fixed in Release 9.0

****

Temporary files are now placed in the directory specified by the the environment variable TMP, or in the current directory if TMP is

not set. [PTM 6682]

****

Compiler Driver - Fixed in Release 8.3

****

The -l switch controls the destination of listing output implied by other listing switches. However, if no other listing switches were specified, an empty listing file was being created. [PTM 5628]

****

On the PC, the driver was creating temporary files in the root directory. This was causing problems for users who have access to their own directory, but not the root directory. Now, the temporary files are created either in the directory designated by the TMP environment variable or in the working directory. [PTM 5685]

****

Compiler Front End - Fixed in Release 9.03

****

The Windows version of the C++ front end now determines the directory to use for temporary files as follows: 1) If environment variable TMPDIR is set, it uses that. 2) Otherwise, if environment variable TMP is set, it used that. 3) Otherwise, it uses the current directory.

Compiler Front End - Fixed in Release 9.02

****

The compiler now conveys correct include file path information to the debugger when in C++ mode.

****

The C++ compiler now uses --noanachronisms as default, as is described in the documentation.

****

C++ static data members are now handled correctly in --anachronisms mode.

****

Compiler Front End - Fixed in Release 9.01

****

A limit on the size of string literals allowed has been increased from 2048 to 64000. [PTM 6700]

****

Compiler Front End - Fixed in Release 9.0

****

The compiler no longer crashes under Windows and NT if an include pathname specified on the command line erroneously begins with a colon. [PTM 6613]

****

Compiler Front End - Fixed in Release 8.5

****

The compiler incorrectly treated volatile bitfields accessed via pointers. They were treated as non-volatile values, which caused the compiler to generate over-optimized code. This is a sample declaration which was not treated right by the compiler:

```
volatile struct {
int f1:3;
int f2:3;
```

```
int f3:2;
} y;
```

[PTM6531]

****

The compiler generated wrong code for divide for short type values because of an incorrect type promotion, when using the compiler option -L. Here is a sample C code:

```
extern unsigned short a,b,c,d;
t() {
a = b * c / d;
}
```

[PTM6602]

****

Compiler Front End - Fixed in Release 8.4

****

Compilations failed with a DOS/16M General Protection Fault when the TMP environment variable was set to a nonexistant directory. [PTM 6378]

****

A compiler syntax error occured when trying to merge multiple string literals in an _ASMLINE statement. Here's an example:

```
#define xxx(p) _ASMLINE(" dc.l _" p "-*")
```

[PTM 6461]

****

The compiler failed in the front end if there were more than 256 characters of command line arguments, exclusive of "-S"

and "-I". [PTM 6469]

****

The front end generated incorrect symbol information for multi-dimensional arrays. [PTM 6438]

****

Compiler Front End - Fixed in Release 8.3

****

A spurious error message was generated for an #endif statement when the -d and -D e1s switches were used together. [PTM 5612]

****

ANSI C allows two string literals to be placed next to each other. This is supposed to be equivalent to using the concatenation. That is,

```
p = "a" "b";
```

is supposed to be the same as

```
p = "ab";
```

Sometimes the concatenated string literal was not formed properly, leading to an invalid string literal and possibly even an invalid object module. [PTM 5636]

****

If a source file had more then 32767 lines, then bad things started happening. Error messages may have referred to negative line numbers, and the optimizer may have produced incorrect results. The limit has been extended to 65536 lines. [PTM 5652]

****

When the -D c1s switch is present, the compiler treats "char" as

equivalent to "signed char." However, when generating a prototype (because of the -mp switch), it uses the word "char" as equivalent to "unsigned char." This can lead to incompatible prototypes. For example,

```
unsigned char f() { ... }
```

gives rise to

```
char f(void);
```

even though (under -D c1s) this is equivalent to

```
signed char f(void);
```

[PTM 5661]

****

Incorrect intermediate language was generated for expressions formed by typecasting a constant to a pointer, dereferencing that pointer, and typecasting the resulting value. For example,

```
(int)(*((char *)1))
```

The incorrect code treated this like the original constant (1), rather than the contents of the byte at address 1. [PTM 5665]

****

By default, the compiler searches for user include files by first looking in the directory containing the source file. This search order can be changed by specifying a null directory name as an argument to the -I switch, e.g., "-I xxx -I -I yyy". This is supposed to search directory xxx, then the directory containing the source, then the directory yyy. The "null directory" construct didn't operate correctly, however. [PTM 5678]

****

The front end did not evaluate the difference between two pointers, even when the pointers being subtracted are both typecasts of integer

constants. This makes such expressions illegal in contexts where compile-time constants are required. For example,

```
static int i = ((char *)10) - ((char *)2);
```

This should be legal and equivalent to

```
static int i = 8;
```

[PTM 5692]

****

The compiler failed to emit a diagnostic and sometimes aborted the compilation when an incomplete enumeration type was used to declare a prototype parameter. For example,

```
f(enum xxx x) { ... }
```

where "xxx" is not defined anywhere in the compilation. [PTM 5702]

****

Incorrect preprocessor output (-E output to a .pp file) was written for "char" and "signed." The use of "signed" caused unpredictable results, possibly including a core dump. The use of "char" printed "unsigned char" (or "signed char," if -D c1s is present). For example, this program:

```
unsigned char x;
```

resulted in this .pp file when compiled under -E:

```
unsigned unsigned char x;
```

[PTM 5710]

****

The compiler gives syntax error messages for unclosed character or string literals. However, it should not emit a diagnostic if these literals occur in source that is excluded by preprocessor

directives, e.g.,

```
#if 0
don't compile this
#endif
```

[PTM 5716]

****

The symbolic debug information associated with two-byte enumerated types was wrong when the -L switch was present. These types were incorrectly described as being four bytes long. This caused the debugger to access such variables incorrectly. Also, the sizeof function delivered the wrong value when applied to variables of such types. [PTM 5783]

****

The compiler incorrectly gave a structure the "contains read-only components" attribute if it contained a pointer to a read-only type. This caused the compiler to reject assignments to the structure. [PTM 5873]

****

The compiler incorrectly gave arrays of pointers to read-only types the "contains read-only components" attribute. This caused the compiler to reject assignments to elements of the array. [PTM 5888]

****

Many users have been puzzled by the error messages for programs like this:

```
int f(char);
int f(x)
        char x;
{ ...
```

According to the rules of ANSI C, this is an error. Either of these examples would be correct, however:

```
int f(int);
int f(x)
        char x;
{ ...
```

or

```
int f(char);
int f(char x)
{ ...
```

The details of the applicable language rules may be found in Section 3.5.4.3 of the ANSI C standard. The basic idea is that when a non-prototype definition is compared with a prototype, the "default promotions" are performed on the parameter types before the comparison.

Since so many people complained about this, we have made this a warning rather than an error. [PTM 5875]

****

The -pack 1 switch did not always work properly when -L was also used. For example, the size of the following structure would come out as 6 bytes instead of 5 bytes:

```
struct s1 {
    char c;
    int i;
};
```

[PTM 5900]

****

There was an obscure bug in the token merge (##) operator. When the right-hand-side of a merge operation is a keyword, the compiler didn't obtain the correct characters for the keyword. In fact, if the left-hand-side was also a keyword, it got the characters for the left-hand side. Otherwise it got "char." [PTM 5939]

****

Compile time conversions of unsigned long constants to float or double was done incorrectly. Instead of doing an unsigned conversion, it did a signed conversion. Therefore, you get the wrong result for constants bigger than 2147483647. For example,

```
double d = (double) 2147483648UL;
```

sets d to -2147493648.0, which is incorrect.

[PTM 6055]

****

The compiler was missing an opportunity for optimization of expressions containing "?:" conditional operators. [PTM 6063]

****

The compiler was missing an opportunity for optimization of expressions of the form "x = !!y". [PTM 6064]

****

The compiler incorrectly calculated the value of a constant left-shift expression (<<). If the value being shifted fit in 8 bits, then the result was incorrectly truncated to 8 bits. [PTM 6065]

****

After a #pragma sep_off, a variable was not correctly treated as being separate under -sd. [PTM 6143]

****

A spurious "constant value truncated" warning was issued on expression with a shift by 0. [PTM 6176]

****

A spurious syntax error was issued for a // comment following

a #define, in cases where there was no empty line after the comment line. [PTM 6231]

****

Compiling a file with listing options caused a fatal syntax error in the optimizer when the filename was in excess of 55 characters. [PTM 6234]

****

Compiler Optimizer - Fixed in Release 9.01

****

2 bugs in the optimizer causing general protection faults under Windows 95 have been fixed. [PTMs 6697, 6698]

****

Compiler Optimizer - Fixed in Release 9.0

****

A problem in the optimizer regarding for loops of the form "for(i=x; i++;) ..." has been fixed. [PTM 6627]

****

-nh now suppresses code hoisting when an address calculation would otherwise be moved. [PTM 6666]

****

A problem in the optimizer regarding while loops of the form "while(i--) ..." has been fixed. [PTM 6665]

****

Compiler Optimizer - Fixed in Release 8.5

****

If _ASMLINE immediately follows the "if" statement, the compiler optimizer incorrectly placed the label that concludes the "if" statement after the assembly instruction resulted from the _ASMLINE. Here is an example:

```
int foo(int a, int b, int c, int d)
{
_ASMLINE(" ori #$0700,SR");
if (a == 1 && b == 1) {
a = 2;
b = 2;
} else if (c == 2 && d == 2) {
c = 1;
d = 1;
}
_ASMLINE(" andi #$F8FF,SR");
return(0);
}
```

[PTM6501]

****

The compiler performed an unsafe optimization by moving conditionally executed loop-invariant operations which may cause exceptions. Loop-invariant operations which may cause exceptions such as divides and pointer dereferences should not be moved.

Here is a sample:

```
{
int a1, a2, i, y, x;
int *p;

x=1;
for ( y = 0, i = 0; i < 10; i++) {
if ( a2 != 0 ) {
        x = (a1 / a2) * i;          /* don't move the divide */
}
y += i * x;
```

```
}
}
```

[PTM6481]

****

A register was not updated properly due to an incorrect strength reduction optimization. This happened in loops where a register is used to allocate an expression depending on the loop counter. See the following example:

```
for (i = 0; i < n; i++) {
        result += ((unsigned long int)b[i]) * ((unsigned long int)b[i]);
        result += a[2*i];
        a[2*i] = result;
        result >>= 8;
        result += a[2*i+1];
        a[2*i+1] = result;
        result >>= 8;
}
```

Here 2*i was allocated in a register by the compiler but was not properly updated.

[PTM6620]

****

The compiler incorrectly identified loops whose "while" condition is TRUE as infinite loops when "break" is hidden within "if" within the loop. As a result, the code following the loop was considered unreachable.

[PTM6634]

****

Compiler Optimizer - Fixed in Release 8.4

****

The optimizer reforms loop limit expressions of the form "!(expr)" into "(expr) == 0". This is done to aid strength reduction which "prefers" the latter condition when transforming loop exit conditions. The optimizer generated invalid intermediate language when the starting expression was of the form "!(x == #)", where # is a constant. [PTM 6432]

****

Compiler Optimizer - Fixed in Release 8.3

****

The compiler incorrectly hoisted a divide inside of an if statement without also hoisting the if statement. [PTM 6401]

Several bugs in the strength reduction optimization code were fixed. Generally, these resulted in correct but sub-optimal code. However, there were a few cases where incorrect code was being generated. It is difficult to characterize the cases where the incorrect code was being generated because they depend upon several independent factors. [PTMs 5616, 5751, 5950, 5992, 6015, and 6271]

****

When a 32-bit integer is converted to single precision floating point type, there is some possibility that truncation will occur, since a single precision floating point only has 23 bits of mantissa (24 bits counting the implied leading 1). When this truncation happens during a conversion of a compile time constant, the compiler should give a warning, but it didn't. [PTM 4228]

****

The optimizer incorrectly optimized away the test in a loop with a continue statement. Here is an example:

```
int doingthis();
void main() {
    int a = -1;
```

```
        while (a == -1) {
                if (doingthis()) {
                    continue;
                }
                a = 0;
        }
    }
```

This was compiled as if the source was simply this:

```
    int doingthis();
    void main() {
        doingthis();
    }
```

[PTM 5610]

****

Sometimes the optimizer does compile-time constant folding of floating point expressions. This was being done in such a way that a good amount of precision was being lost, especially for very small values. [PTM 6003]

****

Under complicated conditions, the optimizer created a situation where a value is read from a register which is not previous assigned. This is a result of a defect in the "code hoisting" optimization. The condition for the bug to occur is as follows:

1) A loop invariant expression appears in a loop which contains a smaller sub-expression.

2) Exactly two uses of the sub-expression appear later on in the same loop,but the sub-expression does not appear previously outside the loop.

3) The second of these two uses can be reached without executing the first use. For example, they may be in opposite sides of an if-then-else.

4) The sub-expression is sufficiently simple that it would be better to compute it twice than to compute it once, store it in memory, and then load it from memory.

5) A register is available over the lifetime of the loop.

If all of these conditions applied, then the generated code would be incorrect. The sub-expression would be computed into a register at the first use, and that same register would be used at the second use. The code would not execute properly if the second use is reached without crossing the first use, since the register would be uninitialized. [PTM 6040]

****

An if condition within the body of a loop was incorrectly removed by the optimizer when a function call was present in the loop. [PTM 6100]

****

The optimizer failed to reassign the correct value number to a folded expression involving a logical OR operator, resulting in a "Bad value number in Constant_Value" fatal error. [PTM 6235]

****

The separate segment and class information of a separate variable was sometimes lost if the first reference to a static variable was not its point of declaration. [PTM 6261]

****

A fatal error was caused by use of a ?: operator. [PTM 6272]

****

Compiler Back End - Fixed in Release 9.02

****

When a switch statement contained a function call with parameters,

and option -j was in effect, under certain conditions a generated conditional branch backwards incorrectly used the short format when the long format was required.

****

Compiler Back End - Fixed in Release 8.5

****

The compiler emitted an internal error when compiling a loop which assigns the same value to consecutive elements of an array. This occurred only when the right-hand side of the assignment is a register value whose type is strictly larger than the array element type. An example follows:

```
extern char x[64];
f(register long c) {
int i;

for (i = 0; i< 63; i++) {
     x[i] = c;
}
}
```

[PTM6573]

****

Compiler Back End - Fixed in Release 8.4

****

Incorrect code was generated for switch statements which have all of the following properties:

1) There are both positive and negative cases labels.
2) There is a case which is labelled by at least three consecutive negative case labels, e.g.,

```
    case -1:
```

```
        case -2:
        case -3: ...
```
3) The switch is relatively "sparse".
4) The switch selector expression is not a register variable.

[PTM 6429]

****

Volatile variables are supposed to receive special treatment by the compiler. Some of this special treatment didn't happen for volatile bitfields.

Any reference to a volatile variable is considered a side-effect, even if the value of the variable is not needed for any apparent purpose. For example, consider this program:

```
volatile int var;
t() {
        var;
}
```

Here the reference to var has no apparent purpose, but since var is volatile, the compiler generates a TST instruction which touches the corresponding memory address. The compiler failed to emit the TST in the case of a volatile bitfield. Note that other optimizations on volatile bitfields were correctly supressed; it's just this "touch on useless reference" process that didn't work. [PTM 6421]

****

The compiler generates the wrong bit pattern for the memory-indirect addressing mode when no index register is present, i.e., ([An]) or (d,[An]). This addressing mode is not generated for CPU32 and 68000-like targets, so this bug only applies to the 68020, 68030, 68040, 68060, and corresponding EC-family targets.

The incorrect opcode seems to execute properly on real 68020 hardware, but may cause problems under software simulation. The problem is that bit number 2 in the full-format extension word is set to 1, when it should be 0. For example, "MOVE ([A0]),D0"

should generate the bit pattern 30300151, but instead generates the bit pattern 30300155.

The assembler processes this addressing mode correctly, so the bug does not occur when the "-ia" switch is present.

[PTM 6410]

****

Compiler Back End - Fixed in Release 8.3

****

Incorrect code may be generated for assignments to or from small structures or doubles. To encounter the problem, all of the following conditions must apply:

1) The target must be a 68000, 68008, 68302 or some equivalent processor.

2) The item being moved must be a field of a larger structure.

3) That larger structure must be inside an array.

4) That array must be indexed by a non-constant array index.

5) That array must be on the stack or accessed via a pointer.

6) The offset from the value in the pointer (or the stack frame) to the start of the array plus the offset of the item within the larger structure must be less than 128 but more than 128 minus the size of the item.

Here is an example that meets all these conditions:

```
struct foo {
    long l1;
    long l2;
} x;
struct bar {
```

```
    char c[124];
    struct bar d;
} *p;
int i;

p[i].d = x;
```

Small aggregate assignments are often done as a sequence of moves with increasing offsets, e.g.,

```
MOVE.L source,dest
MOVE.L source+4,dest+4
...
```

The problem happens when the source or destination address has the form

```
c(An,Dm)
```

where the constant c is less than 127. In this case the compiler chooses the "d8(An,Dn)" addressing mode. Increasing the offset can cause the total displacement to exceed 127, at which point the addressing mode is no longer valid on a 68000 processor. [PTM 5600]

****

The compiler generated code which could cause an address exception for statements of the form

```
strcpy(p,"....");
```

where .... is a string literal, and p is a stack-resident local variable. This bug affected only the 68000, 68010, 68302, 68332, and 68340 targets. [PTM 5630]

****

After a procedure call, the caller must pop the arguments off the stack. The compiler delays this fixup in the hope that it may be able to combine several call fixups into one operation. However, the fixup was getting lost entirely if an inline asm call appeared when a non-zero fixup was pending. As a result, asm macros that affect the

stack weren't working correctly. [PTM 5675]

****

The compiler omits TST instructions if the condition codes have already been properly set by previously emitted instructions. When a label is emitted, this information is flushed. This information must also be flushed when an in-line asm macro is invoked, but this was not done properly. As a result, TST instructions were missing after in-line asm macro calls. [PTM 5682]

****

The compiler uses several clever sequences to avoid multiplications by constants. The particular sequence chosen depends on the value of the constant. There was a bug in one such sequence when the constant was negative. The bug was that the negative sign got lost, so instead of multiplying by "-N" it multiplied by "+N". One number which leads to this error is -100. [PTM 5744]

****

When two pointers are subtracted, the difference in bytes between the two pointer variables must be divided by the size of the element to which the pointer points. For example, if you have two pointers to 4-byte integers, and one contains 0 and the other contains 4, then the difference between these two pointers is 1, not 4.

When the optimizer makes a CSE out of the constant which scales the difference between two pointers, then the back end can get a fatal syntax error. The error only occurs if the pointer subtraction is the first use of the constant. [PTM 5785]

****

The compiler keeps track of the values in variables in an attempt to avoid redundant loads and stores. Naturally, this information must take into account the effects of assignments, especially when unions are involved. That is, an assignment to one element of a union must invalidate any information pertaining to other union elements which occupy the same or overlapping storage. The compiler did not always correctly invalidate its internal information at bitfield assignments.

As a result, incorrect code was sometimes generated for unions which overlay bitfields with other elements. Here is an example:

```
unsigned int x;
ptm5898() {
    union {
        struct {
            unsigned int ch1 : 14;
            unsigned int ch0 : 2;
        } Bits;
        unsigned int Word;
    } unionvar;

    unionvar.Word = 0;
    unionvar.Bits.ch0 = 1;
    x = unionvar.Word;
}
```

Here the "Bits" element and the "Word" element overlay one another. After the second assignment, "unionvar.Word" is no longer equal to 0, but the compiler thought it was. Therefore, it stored 0 into x. [PTM 5898]

****

The compiler optimizes away the LINK instruction in procedures that don't appear to need the A6 register. (This optimization can be suppressed with the -nl command line switch.) The compiler missed one case where A6 actually is needed, and thereby created invalid code. That is, it uses A6, but has no LINK instruction to set up A6.

The case that was missed is a procedure which has a parameter which is used as the argument in an _ASM macro invocation. For example:

```
t(int x) {
    macro(x);
}
```

where "macro" is defined elsewhere as an _ASM macro. [PTM 5930]

****

The compiler used the wrong register after a pointer post-increment. Here is a small example:

```
extern char *f();
extern int i;
extern char *q;
ptm5979() {
    char *p;
    p = f();
    *p++ = 'a';
    i = (p-q);
}
```

Here the local variable "p" is allocated in the A1 register. After the procedure call, the value in "p" is present in both A0 (where "f" returned its value) and A1. After the post-increment, this is no longer true, but the compiler behaved as if it were. It therefore subtracted "q" into A0 to compute the "p-q" expression, but that is invalid. [PTM 5979]

****

Under hardware floating point, a compare between a post-increment register variable of type float or double with a floating constant may have resulted in incorrect code being generated. The problem was that the bits of the floating constant were computed incorrectly. This problem only occurred under hardware floating point. [PTM 6052]

****

The compiler generated bad code for an aggregate assignment where the left-hand side is "*p++" and "p" is a register variable. The post-increment was never performed. This kind of situation can also arise through optimization, notably strength reduction. For example, this kind of code would be affected, as long as a[i] has an aggregate type:

```
for (i = 0; i < 10; i++) {
    a[i] = b[i];
}
```

[PTM 6087]

****

The compiler generated an invalid addressing mode for a structure field reference under the following conditions:

1) The byte offset from the structure to the field is bigger than 127.
2) The structure is in A5-relative memory.
3) Previous code generation has caused the A5-relative offset of the structure to be loaded into a data register.

[PTM 6144]

****

Incorrect code was generated for some rather complicated expressions involving conditional operators. To have a chance to hit the bug, the expression must have the following form:

x = y <operator> (a ? b : c);

Here "x" must be a variable packed to a register which not packed to any variables that appear on anywhere within the right hand side of the assignment. Both "y" and either "b" or "c" must be expressions involving more than one operand. "<operator>" must be "&", "|", "^", "/", "%" or "-". [PTM 6162]

****

A logical and (&) operator which is used in a boolean context can sometimes be implemented by a BTST instruction. To do this, one of the operands of the "&" must be a constant which is a power of two. The compiler sometimes generated incorrect code when the other operand was a post-incremented register variable or a memory location addressed using post-incremented registers. For example, "if (i++ & 1)" or "if (a[i++] & 1)", where "i" is packed to a register. [PTM 6165]

****

Assembler - Fixed in Release 8.3

****

The default qualifier for the FEQU directive should be .S, not .W. [PTM 5417]

****

The assembler did not accept a COMMON directive which does not have an ABSOLUTE specification but does have a class name. For example,

```
COMMON sect1,,"cclass"
```

incorrectly resulted in a syntax error because only one comma was expected. [PTM 5626]

****

The line number information within assembler listing files was incorrect after a macro call that resulted in the MEXIT directive being executed. [PTM 5664]

****

Any comment on a macro invocation line yielded a warning message. [PTM 5680]

****

The NARG symbol was being set to 1 when a macro was called with no arguments. Now it is set to 0. [PTM 5863]

****

A dangling comma on a list of XDEF or XREF symbols caused a General Protection Fault in the assembler. [PTM 6033]

****

Linking Locator - Fixed in Release 9.03

****

When linking C++ programs, spurious warnings about unresolved references are no longer emitted.

****

Linking Locator - Fixed in Release 9.02

****

When linking C++ programs, spurious warnings about unresolved references to _dtors and _ctors are no longer emitted.

****

Linking Locator - Fixed in Release 9.0

****

llink now supports the -opfile <file> command-line option, where <file> contains an arbitrary number of filenames and linker options. [PTM 6639]

****

A problem with llink not deleting temporary files has been corrected. [PTM 6684]

****

Linking Locator - Fixed in Release 8.3

****

The linking locator required that all RESERVE commands appear before any LOCATE commands. This restriction was imposed so that the linking locator could prevent a LOCATE command from placing a segment in a region named in a subsequent RESERVE command. Now, the linking locator will accept RESERVE commands that follow LOCATE commands, as long as the RESERVE areas are still available. [PTM 5627]

****

When a program was missing the extern before the declaration of myromp, the linking locator failed to properly recognize this error. Now, the following fatal error is generated:

ROMP output segment '_myromp' clashes with existing symbol

[PTM 5924]

****

On the PC, the linking locator did not correctly handle absolute pathnames of modules in library index files. This meant that you couldn't link with such a library index file unless you performed the link in the same directory as the index file. [PTM 5672]

****

The linking locator failed to issue a diagnostic if it was unable to satisfy a locator command of the form:

LOCATE ({usep}: address-range);

[PTM 5938]

****

On the PC, a General Protection Fault could occur when the files being linked contained a very large number of symbols. [PTM 6037]

****

C Run-Time Libraries - Fixed in Release 9.03

****

string.h and time.h now correctly typedef size_t to be unsigned long when compiling for C++.

C Run-Time Libraries - Fixed in Release 9.01

****

Float to unsigned long conversions were not happening properly if the original float value was between 2^31-1 and 2^32-1 (between the max values of signed long and unsigned long). Values in this range were always converted to be the max value for signed long. [PTM 6701]

****

C Run-Time Libraries - Fixed in Release 9.0

****

Library function itob is now reentrant. [PTM 6695]

****

Problems with stdarg.h and data smaller than ints have been resolved. [PTM 6679]

****

printf now handles "%- " format correctly for very small values. [PTM 6646]

****

C Run-Time Libraries - Fixed in Release 8.5

****

InterTools did not load the Motorola Numeric Floating Processor (nfp) package. This problem was significant for the 68040 and 68060 series only.

[PTM6642, PTM6643]

****

C Run-time Library - Fixed in Release 8.3

****

Previously, the U.S. Software floating point run-time library modules did not support 32-bit addressing off of register A5. This was a problem for users of the -b5 (use 32-bit A5-relative offsets) compiler switch. Now, however, "big-A5" versions of these run-time library modules are supplied.

When using the -b5 compiler switch, it is necessary to rebuild the run-time library so that global data is referenced properly, given that A5-relative offsets are no longer limited to 16 bits. This can almost be accomplished by recompiling all the .c files in the library with -b5. However, there are a few assembly language modules which also use A5-relative addressing. To make a "big-A5" library, alternative versions of these modules must be used. Here are the alternative versions which must be used. This is analogous to the procedure for building a "no-A5" library; consult the Building Libraries That Do Not Use A5 application note in the InterTools User's Manual for more details about how to do that.

Software Floating Point Libraries

Without -L:		With -L:	
Normal:	Big-A5:	Normal:	Big-A5:
adln.ln	adlnb.ln	adlnl.ln	adlnlb.ln
adlog.ln	adlogb.ln	adlogl.ln	adloglb.ln
adsqrt.ln	adsqrtb.ln	adsqrtl.ln	adsqrtlb.ln
dpopns.ln	dpopnsb.ln	dpopns.ln	dpopnsb.ln
dpfncs.ln	dpfncsb.ln	dpfncs.ln	dpfncsb.ln
fpopns.ln	fpopnsb.ln	fpopns.ln	fpopnsb.ln
xlfncs.ln	xlfncsb.ln	xlfncs.ln	xlfncsb.ln

Hardware Floating Point Libraries

Without -L:		With -L:	
Normal:	Big-A5:	Normal:	Big-A5:

acos.ln	acosb.ln	acosl.ln	acoslb.ln
asin.ln	asinb.ln	asinl.ln	asinlb.ln
log.ln	logb.ln	logl.ln	loglb.ln
log2.ln	log2b.ln	log2l.ln	log2lb.ln
log10.ln	log10b.ln	log10l.ln	log10lb.ln
sqrt.ln	sqrtb.ln	sqrtl.ln	sqrtlb.ln

[PTM 5622]

****

Several assembly language members of the run-time library were not properly made position-independent. This causes problems when linking with code compiled under the -ps or -pc switches. The affected library modules were:

1) Several routines used to support software floating point.

2) The "jmp to self" in the exit routine.

3) The "jsr to main" in the pmain routine.

4) Out-of-line support for unsigned 32-bit integer divide and modulus, (used only with the 68000, 68010, and 68302 targets).

5) The "div" and "div" library routines.

[PTM 5640]

****

Unsigned 32-bit modulus (the C "%" operator) is computed out-of-line for processors that do not have a real 32-bit by 32-bit divide instruction. In particular, the 68000, 68010, and 68302 all fit this description. The library routine which computes the unsigned 32-bit modulus, _wmod, operated incorrectly if the divisor were bigger than the maximum signed 32-bit integer. In that case, it truncated the divisor to 16 bits, then computed the 32-bit by 16-bit modulus. This was not only incorrect, but could result in a division by zero.
[PTM 5954]

****

The "precision" option on unsigned integer printf formats padded with zeros on the wrong side. For example, this:

```
printf("%.3u",1);
```

should print "001", but instead printed "100". [PTM 6012]

****

The %g printf option sometimes produced incomplete results. [PTM 6027]

****

The 16-bit integer version of the "div" library routine delivered incorrect results if the numerator argument was negative. [PTM 6099]

****

Formatter - Fixed in Release 9.02

****

Certain uses of form with -x, and with "-d -f c" no longer cause a general protection fault.

****

The use of option -x with form695 has always been illegal and is now flagged as such by the utility.

****

Formatter - Fixed in Release 9.0

****

P&E debugging format is now supported correctly. [PTMs 6605 & 6688]

****

A problem which caused form to crash in certain situations has been corrected. [PTM 6633]

****

Structure offsets greater than 32767 are now handled correctly. [PTM 6673]

****

Formatter - Fixed in Release 8.5

****

form695 produced a segmentation violation on a large test case when using -d when an assembler module had no code.

[PTM6611]

****

Formatter - Fixed in Release 8.4

****

The form program had a table of record field symbol entries which was limited to 400 elements. The limit was not correctly checked for overflow and thus caused the program to crash when a structure contained more than 400 fields. The record field symbol entries are now chained and allocated from the heap when needed.

[PTM 6437]

****

Formatter - Fixed in Release 8.3

****

Several different bugs in the COFF output format have been fixed.
[PTMs 5618, 5641, 5706, 5745, 5851, 5882, 5894, and 6069]

****

Line information was incorrectly generated for functions following
an include file which contained code. [PTM 6153]

****

@(#)m/r/c682u/readme 1.20