

Bug Identifier	Problem/Bug Description	Severity	Workaround Description	Affected Releases	Fixed Release
V1.00A-1	Re-definition of a struct or union causes a crash in ETEC_cc.exe. E.g. struct XC { int a; }; struct XC { char b; };	3	The problem is triggered by an error in the source code. Correcting the source code fixes the problem.	V1.00B and earlier	V1.00C
V1.00B-1	It has been found that the Byte Craft code development system treats the type "char" as unsigned, while ETEC treats "char" as signed. Both implementations are valid according to the C99 language specification. However, for improved compatibility, ETEC will change its implementation to default "char" to unsigned.	4	Change variables declared to have type "char" to instead of type "unsigned char", or use the typecast "(unsigned char)" where appropriate.	V1.00B and earlier	V1.00C
V1.00B-2	ETEC is not properly performing the integer promotions on the operands of a shift expression. When the left-hand operand is 8 bits and shifted to the left, bits to the left of the low 8 bits (up to 16 more bits) should be preserved if the destination type is larger than 8 bits. E.g.: char x = 1; int y = (x << 8); // y should end with a value of 256, not 0	2	For now, typecast the 8 (or 16) bit operand of the shift to the destination type before the shift, e.g. char x = 1; int y = (((int)x) << 8); // y always gets 256	V1.00B and earlier	V1.00C
V1.00B-3	Several minor issues were found in the map & analysis files with regards to register variables and static variables.	3	No workaround, but these output files provide extra information and are not considered to be at the same criticality as the executable image, etc.	V1.00B and earlier	V1.00C
V1.00B-4	Channel frame _Bool bit arrays are getting listed under the 24-bit types in the _defines file, and thus getting the wrong auto-generated macro name.	3	The macro can be used as-is because the offset is correct; however the macro name is wrong and will change (be corrected) in future versions.	V1.00B and earlier	V1.00C
V1.00B-5	Indirect volatile accesses may not be treated as volatile. E.g. volatile int* x_ptr = &g_int; *x_ptr = 5; int y = *x_ptr; The second access to *x_ptr may not be preserved.	2	If possible, convert indirect volatile accesses to a direct volatile access. Otherwise, no work-around.	V1.00B and earlier	V1.00C

V1.00B-6	The ETEC_cc.exe options -optDis and -optEn (optimization disable/enable) is not working properly for most optimization types; the option is ignored rather than processed.	3	The work-around is to use the matching #pragma disable_optimization <optId> in the source file instead of at the command line. Note that enabling/disabling of individual optimizations should not be done in general.	V1.00B and earlier	V1.00C
V1.00B-7	Bitfield bit sizes were being output incorrectly for channel frame structures in the defines file; the bit offset for the field rather than the size was showing up in the output.	3	Use user-defined macros instead of depending upon the defines file output.	V1.00B and earlier	V1.00C
V1.00B-8	In some cases, referencing a variable of an array type is not being treated (converted) as (to) the appropriate pointer type in expressions, resulting in a compilation error in cases that are not in error.	3	Use an explicit typecast to convert the array reference to the appropriate pointer type. E.g. int24_array[10]; int24* arr_ref = (int24*)arr_ref + 5;	V1.00B and earlier	V1.00C
V1.00C-1	Certain kinds of syntax errors can cause "trickle-down" errors as the compiler tries to recover, sometimes resulting in a crash or hang.	3	Fix the syntax errors that start the problem.	V1.00C and earlier	V1.00D
V1.00C-2	In some particular circumstances, code can be generated for multi-bit bit shifts that results in corruption of a in-use register.	2	Should the problem be encountered (it is rare), try breaking up the bit-shift into small pieces.	V1.00C and earlier	V1.00D
V1.00C-3	De-referencing a pointer that is kept as a register variable can in some cases result in de-allocation of that register and thus re-use of it, resulting in corruption.	2	Should this problem be encountered, the work-around is to cast the pointer to an int, then back to the correct pointer type. int8* c_ptr; *((int8*)((int)c_ptr)) = 7;	V1.00C and earlier	V1.00D
V1.00D-1	If an expression included signed division (or signed modulus) and the division op was not the last of the expression, the division result could get overwritten.	2	Break up expressions so that signed division (modulus) is not below any other operations besides assignment.	V1.00D and earlier	V1.10A
V1.00D-2	When shifting a local variable by a variable number of bits (e.g. var1 <<= var2;), the local variable var1 can later be corrupted.	2	Place the variable to be shifted in a temporary first. { int temp = var1; var1 = temp << var2; }	V1.00D and earlier	V1.10A

V1.00D-3	Making an array reference from a pointer, where the pointer is itself the product of an expression, is resulting in the pointer value rather than the array element. E.g. <code>int A[10]; int b = (A+4)[1];</code> // b should get the 6th element of A	2	Do the pointer arithmetic in a separate expression: <code>int* c = A + 4; b = c[1];</code>	V1.00D and earlier	V1.10A
V1.00D-4	Arguments to signed division or signed modulus that are also stored in registers are having the absolute value operation applied as a side-effect. Variables stored in RAM are not affected. E.g. <code>int a = 7, b = -7, c; c = a / b;</code> // b getting set to 7 if it was in a register	2	Place the problem variable in a temporary before the operation: <code>{ int temp = b; c = a / temp; }</code>	V1.00D and earlier	V1.10A
V1.00D-5	When the sizeof operator is applied to a constant the wrong size may result, e.g. <code>sizeof(1)</code> may result in "1" rather than "3".	2	Take the sizeof of the desired type instead: <code>sizeof(int)</code>	All versions	TBD
V1.00D6	A constant value in a <code>switch()</code> expression is not compiling correctly. E.g. <code>switch (5) { ... }</code>	3	This is an unnecessary construct; avoid it.	V1.00D and earlier	V1.10A
V1.00D7	It appears that ETEC integer promotion rules are not correct in all cases. For example, the code <code>{ unsigned char a = 1; unsigned char b = 2; int c = a - b; }</code> should yield a value of -1 in c but instead ETEC-generated code results in 255.	2	Cases such as the example shown can be corrected through the use of explicit typecasts, e.g. <code>int c = (int)a - (int)b;</code>	All versions	TBD

Bug Severity Level Descriptions:

- 1 – Problem causes complete work stoppage. No work-around is possible. The problem is likely to be hit by most users. This level of bug will typically trigger a new release or patch in a short time frame.
- 2 – A difficult problem to track down, such as incorrectly generated code. Typically there is a work-around available for this kind of bug.
- 3 – A bug that is easy to spot, and/or generally has a straight-forward work-around, or has minimal impact.
- 4 – Not truly a bug (i.e. tool is within spec.), but rather something that might affect compatibility or usability. Work-arounds available.