Known Bugs in ETEC Version 2.30

Bug Identifier	Source	Problem/Bug Description	Severity	Workaround Description	Affected Releases	Fixed Release
V1.00D-5 (2009-Dec-15)	internal	When the sizeof operator is applied to a constant the wrong size may result, e.g. sizeof(1) may result in "1" rather than the expected "3" bytes.	2	Take the sizeof the desired type instead: sizeof(int)	All versions	TBD
V1.20A-14 (2009-May-20)	internal	Chan interrupt opcodes may be moved relative to adjacent RAM instructions by the optimizer. This may cause unexpected results, particularly in the case of a DMA interrupt.	3	Use _OptimizationBoundaryAll() or #pragma opimization_boundary_all if there is concern that an interrupt may cross a critical RAM access.	All versions	TBD
V1.25A-11 (2009-Sep-28)	internal	If pointer arithmetic generates a negative result, and the object pointed to is larger than 1 byte in size, ETEC code will generate an incorrect result. This is because an unsigned shift (or unsigned divide) is applied after the pointer arithmetic to convert from byte addressing to object indexing.	3	Keep pointer arithmetic results in the non-negative domain.	All versions	TBD
V1.25B-6 (2009-Dec-9)	internal	The _STACK_SIZE_ defines macro gets the calculated value of the worst-case stack depth. In certain rare cases, this value can be slightly larger than the actual worst-case. This can occur when a stack usage of a register save and restore (e.g. in a called C function) is eliminated via optimization. Such a register save requires 4 bytes of stack space, but the removal of it is not currently getting accounted for in the stack size calculation.	4	Care should be taken in that in some rare cases, a _STACK_SIZE_ value that is non-zero can still mean that no stack is actually utilized. Another way to verify that no stack is used is to make sure that no <func/class name>__STACKBASE_ macros are defined.	All versions	TBD

V1.25B-7 (2009-Dec-11)	internal & customer	The optimizer/analyzer does not yet support reentrant functions, whether they be callable C functions or ETEC code fragments. Reentrance is supposed to be detected and cause an error, but in some cases this detection failed, allowing for optimization to continue. Sometimes the result could be a linker crash, or sometimes invalid code generation, or in some cases working code resulted.	3	Avoid writing reentrant functions until the ETEC optimizer/analyzer fully supports them.	All versions	V1.25C (reentrance detection), TBD (support reentrance)
V2.23B-5 (2014-Mar-18)	internal	In some cases when making a fragment call, and the fragment is contiguous with the calling code (i.e. jump can be eliminated), the link-time optimizer mistakenly optimizes out code it should not.	2	This situation, if encountered, can be corrected by re-arranging the code to prevent the fragment call and fragment code from being continguous.	V2.00A and newer	TBD
V2.23B-7 (2014-Jun-6)	customer	The C preprocessor is currently allowing the same macro to be expanded in multiple replacement passes, which causes the preprocessor to break when such "recursion" is encountered.	3	Avoid self-referencing preprocessor macros.	All versions	TBD
V2.30A-1 (2014-Jun-27)	internal	When an output path/file is specified that is different than the source file directory when compiling a source file that contains inline assembly, the inline assembly debug information does not come through correctly nor does the listing file generate correctly. The code does compile correctly.	3	Use same directory, or ignore the issue.	All versions	V2.30B
V2.30B-1 (2014-Jul-22)	customer	The 32-bit fract type is documented, but was actually unsupported. Like the 32-bit int type, the only allowed operations are load/store.	4	Use the 32-bit int type if wanting to load/store 32 bits.	All versions	V2.30C

V2.30B-2 (2014-Jul-30)	customer	The register re-use optimization can produce buggy executable in the case where an 'if' statement (not 'if-else') contains a call to fragment, return, or break under the 'if'. This case may lead to falsely determining that two variables usage does not overlap, allowing them incorrectly to utilize the same register.	2	There are two ways to work-around this problem. One is to disable the optimization by using the compiler option "-optDis=0x27". The second is to create a dummy else condition:
				if (x)
				{
				// ...
				return;
				}
				else // work-around bug V2.30B-2
				;
					V2.30A and newer	V2.30C

V2.30B-3 (2014-Jul-31)	customer	Overflow of global/engine scratchpad memory is not being detected properly, which can lead to overlapping memory usage and execution bugs. The bugs only occur if global memory (1024 bytes) or engine memory (512 bytes) is exceeded by scratchpad memory at the start of the linking stage (note that link-time optimizations can compress scratchpad useage). One way the problem can be detected is by looking at the map file .globalscratchpad/.enginescratchpad segment - addresses for same-size objects should always be ascending, but if they descend at any point in the ordered list, this bug may have been activated.	2	The only work-around is to reduce global (or engine) data and scratchpad memory usage to below the limit. This can be done by moving items to channel frame memory, moving functions inline, or other techniques.	All versions	V2.30C

V2.30B-4 (2014-Aug-11)	customer	The compiler executable (ETEC_cc.exe) is incorrectly checking the validity of any search paths passed to it, which may lead to a warning and having the search path filtered out (which then leads to a compile failure when the needed include file cannot be located). The search path is being checked against the current working directory rather than the source file directory. When C preprocessing is done via ETEC_cpp.exe, the correct seacrh path is checked. The invalid and uneeded ETEC_cc.exe check will be removed.	3	There are multiple workarounds. One is to create an empty directory to allow the incorrect ETEC_cc.exe check to pass. Another is to change the build process so that the current working directory and source file directory are the same when building.	V2.30B	V2.30C
V2.30B-5 (2014-Aug-11)	customer	When a pragma write is outputting the value of an enumeration literal, and the enumeration type is defined in multiple object files that are being linked, the literal value gets replicated resulting in an incorrectly generated pragma write file.	3	No simple work-around for pragma write files, but the problem does not exist in the auto-defines host interface file.	All versions	V2.30C

V2.30D-1 (2014-Nov-26)	customer	The register re-use optimization can produce incorrect executable code in two cases. In one case, if two local variables are used in the same expression but do not otherwise overlap in usage, they may incorrectly get assigned to the same register. The second case can occur when one variable is declared at a higher scope, and then is accessed both before and after an inner scope variable, which may incorrectly get assigned to the same register.	2	There are generally work-arounds for any of these types of problems encountered. One way to work-around an issue is declare the conflicting local variables in the same scope. A second way may be to reverse operands in an expression to make the optimization think there is an overlap. For example, in the following code x and y are presenting the conflict bug:
				t = x + 1;
				if (x > y) ...
				Can be changed to:
				t = x + 1;
				if (y < x) ...
				in order to force the determination of an x/y overlap. Detection of both variables in use simultaneously in an expression is in the current algorithm, but the check was not getting activated in all the necessary cases. A third way is to disable the register re-use optimization by passing the "-optDis=0x27" option to the compiler.
					V2.30A and newer	V2.31A

V2.30D-2 (2015-Jan-7)	customer	The duplicate expression optimization can cause problems in certain specific cases where the detected duplicate expression gets converted under the hood to a test opcode. The most likely case for this to occur is when a single bit of a variable/expression is being tested using a bitwise and operator.	3	Disable the duplicate expression optimization in the compilation of the problem file by specifying "-optdis=0x21" on the command line.	V2.00A and newer	V2.31A

V2.30D-3 (2015-Jan-22)	customer	If state enumerations (ETEC state switch extension) are used in multiple translation units (separately compiled C files) and have enumeration literals that share exactly the same name, the literals can resolve incorrectly at link time.	3	Make sure all state enumeration literals have unique names across all source code to be compiled into an executable (unique to all regular and state enumerations). The fix to this bug will be to make sure such diuplicate cases are caught and flagged as an error.	All versions with state enumeration feature	V2.31A
V2.30D-4 (2015-Apr-1)	customer	In some cases using relational operations with one operand being a constant 0 are resulting in code being generated that is not correct for all inputs. This happens when the variable ('x') operand is signed and is at the maximum negative value, and the expression is one of the following: (x <= 0), (0 >= x), (x > 0), (0 < x).	2	The problem can be worked around in general by first putting the constant value 0 in a temporary variable, and then using that variable in the relational expression.	All versions	V2.31C
V2.30D-5 (2015-Apr-14)	customer	The linker can hang in certain unusual cases with loops and threads calling threads.	3	The problem may be able to be worked around by disabling optimizations over sections of code.	All versions	V2.40A
V2.30D-6 (2015-Jun-9)	internal	Compilation of signed division can under certain ciurcumstances fail due to a shortage of registers.	3	Eliminate use of inefficient signed division.	All versions	V2.40A

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.