Known Bugs in ETEC Version 2.42

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
V2.42A-1 (2016-Feb-19)	customer	A case has been seen where a memory location (variable) is written twice in the code, with an intervening read of the variable inside a loop, the initial write gets optimized away mistakenly. This leads to an incorrect value being read in the intervening loop.	2	Use "#pragma optimization_boundary_all" to block the problem from occurring.	All versions	V2.42D

V2.42A-2 (2016-Apr-07)	customer	There is a case where when a branch (if) is taken based upon a channel hardware state, and both branches first action is to change the chan register in the same manner, that opimization may incorrectly move than chan change before the channel hardware state query. E.g.
		`if (channel.PSTI == 1)`
		`{`
		`  chan += 1;`
		`  // ...`
		`}`
		`else`
		`{`
		`  chan += 1;`
		`}`
		In general, the problem is that in rare cases, a chan register update can cross a test of input or output pin states.
			3	Use "#pragma optimization_boundary_all" to block the problem from occurring.	All versions	V2.42D

V2.42A-3 (2016-Apr-25)	internal	If a duplicate expression optimization ends with a function call parameter, the register used for the optimization can be left marked as in-use, and thus can lead to a running out of registers error downstream (compile error).	3	Disable the duplicate expression optimization in the compiler with the "-optDis=0x20" option.	V2.00A and newer	V2.42D
V2.42J-1 (2016-Aug-15)	internal	_Bool bit array initializers are not parsing correctly, leading to incorrect initial values when such initializers were used.	3	Do not use _Bool bit array initializers; initialize at runtime.	All versions	V2.43A

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.