Known Bugs in ETEC Version 2.00

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)

V2.00A-1 (2011-May-11)	customer	When directly accessing the mach/macl registers after an operation, it is recommended the operation in question be done with an intrinsic function to ensure the user-expected MAC/MDU operation is used. For example, if the following code is written assuming that the MDU is used:
		x = y * z;
		result = mach;
		The user may or may not get the intended code generated. If either of the y or z parameters are actually a constant and a multiple of 2, then the compiler may choose to generate the operation using bit shifting for tighter or faster code. By using an intrinsic, the user guarantees the desired hardware function is used. Unfortunately, some MAC/MDU operations are not yet covered by intrinsics - __mults, __multu and __divu intrinsic functions will be added in the next release to provide full support. Then the example above should be written as:
		__mults24(y, z);
		result = mach;
			4	When a multiply hardware operation is required, it is best to put a constant parameter into a variable or register to guarantee the desired opcode is generated (until the proper intrinsic function is available).	All versions (2.00A more so)	V2.00B

V2.00A-2 (2011-Jun-1)	customer	Declaring prototypes for eTPU-C functions (as designated by the #pragma ETPU_function) can cause symbol type conflicts in the linker. Note that once this is fixed, it is important that any prototype declarations and the actual function definition come follow the #pragma ETPU_function.	3	The work-around is to not declare prototypes of eTPU-C functions - they cannot be called by another function anyways.	All versions	V2.00B
V2.00A-3 (2011-May-25)	customer	Under certain conditions (e.g. scratchpad programming model, callable C functions), there can be cases where complex expressions cause the compiler to run out of temporary registers, resulting in a compilation failure. Going forward the compiler will be modified to use better register allocation techniques. In scratchpad mode, rather than error when no temporary registers are available, the compiler will attempt to save off a register value to scratchpad on a temporary basis so that the register can be used in expression processing.	3	When a compilation fails due to running out of registers, you can explicitly tell the compiler to to use one less register for holding variables by specifying "-optDis=0x10". However, this can affect overall optimization and thus is generally not the best solution. Re-arranging the source code slightly can often overcome the problem - in particular only declaring local variables where they are needed, particularly in sub-scopes (late declaration) can be helpful.	All versions	V2.00B
V2.00B-1 (2011-Aug-17)	customer	When the const qualifier is combined with an enumerated type in a function parameter, the compiler will falsely declare there is a type mismatch between prototypes/defintions of the function.	3	For this particular case, do not use the const qualifier.	All versions	V2.01A
V2.00B-2 (2011-Aug-17)	customer	When using the #pragma export_autodef_macro and #pragma export_autodef_text, multiple compiled instances of the export are not being detected (e.g. when the export is in a multiply included header file) and filtered.	3	Use this pragma from a once included / once compiled .c file to avoid multiple instances.	All versions	V2.01A

V2.00B-3 (2011-Sep-1)	customer	In some cases a duplicate expression optimization is being applied where it should not. It is related to expressions involving structure members. For example, in some cases a series of code such as
		x = A.member1 * B.member1;
		y = A.member1 * B.member2;
		Could end up getting compiled as
		y = x = A.member1 * B.member1;
			2	Use the optimization_boundary_all pragma between the expressions that are being identified as duplicates.	V2.00A-B	V2.01A

V2.00B-4 (2011-Sep-1)	customer	When a structure is declared that includes just two members, one of size 16 bits and one of size 8 bits, the members are being accessed incorrectly for read and write operations.	2	Using bitfields works around the problem, or adding a dummy member.	All versions	V2.01A
V2.00B-4 (2011-Sep-29)	customer	When elements of an array are of struct type, where the the struct is larger than 4 bytes (one memory access) in size, there are cases where an assignment (copy) generates bad code. For example, assuming the elements of the arrays below are structs (or unions) of size greater than 4 bytes, the code will fail to compile correctly. struct_array_dest[i] = struct_array_src[i];	2	One way to work around this struct copy issue is to copy them member by member. Note that in general it is inefficient to do struct copies and they should be avoided if possible.	All versions	V2.01A
V2.00B-5 (2011-Oct-21)	customer	The beta auto-struct feature does not properly pad out the host interface structures in some cases, resulting in incorrect structure layouts and sizes. The primary case where this could happen is when "fastaccess" data packing is in use. In addition, arrays of elements of struct/union type are not supported (before this fix), but when they are encountered an error is thrown rather than an incorrect auto-struct generation.	3	Avoid arrays in fastaccess mode, accept arrays fo 24-bit or 32-bit elements. This is not a very viable workaround. V2.01A fixes the issues highlighted in this bug report.	All versions	V2.01A

V2.00B-6 (2011-Nov-29)	internal	#pragma placement within code can result in incorrect code generation. The failure case detected involved a #pragma placed right after an "if () {}" statement (no 'else' clause). Some #pragmas are "code pragmas" in that they affect code generation and thus syntactically act very much like a C statement - these type of pragmas must be placed in the code like C statements. Going forward, the pragmas that are "code pragmas" are documented in the reference manual.	3	As can be see from the bug description, the work-around is often to place code above the #pragma line inside a compound statement - { }. Cases like that described in the bug report - "if () {}" - have been fixed.		V2.01A

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.