Title: Disposition of Comments received on the PDTR 18037 ballot (SC22 N3470, a.k.a. WG14 N979)
Source: Embedded-C subgroup of WG14 meeting in Santa Cruz, 10-14 October 2002

Comments from Germany

GER-1: The concept for fixed-point arithmetic is tailored to digital signal processing. An extension to support a greater number of fixed-point types may pose some difficulties. Other application areas might require decimal fixed-point numbers (as opposed to binary fixed-point numbers) and a different handling of overflow and rounding.

Response: WG14 accepts the German comment as highlighting an area that will possibly need future consideration. However it does not consider it relevant in the context of the part of the current Technical Report concerned with providing facilities for embedded systems.
We hope that the publication of this TR will enable those concerned with other requirements for fixed-point arithmetic types to consider how those might be best provided along side those specified in this TR.

Comments from The Netherlands

The Netherlands approves the document with the following remarks. None of the points mentioned are considered to be critical at this point in time.

NL-1 Section 2.1.4 does not define what the result is when an integer type is combined with a fixed-point type and one of the types is signed and the other is unsigned (e.g., signed fract * unsigned int). We assume that this also yields a signed result.

Response: Accepted: the result type of a combination of an integer type and a fixed-point type is the fixed-point type (rule 2 of 2.1.4); rule 3 does not apply in this case; text will be improved.

NL-2 The operation 'int + fract' (or 'int - fract') is for almost all values of the int operand a useless (and possibly erroneous) operation. The Rationale should make clear why this combination is still allowed (should an implementation be encouraged to flag these operations?).

Response: Accepted; text will encourage to make this a warning, but still allow it.

NL-3 Add library functions for 'int * fract -> int'. Rounding towards zero?

Response: Accepted.

NL-4 Add library functions for 'int / fract -> int'. Rounding towards zero?

Response: Accepted.

NL-5 The last paragraph of section 6.5 of the C standard introduces the concept of a `contracted expression' that allows implementations to enhance the speed of the implementation while sacrificing predictability, and perhaps (almost as a side-effect?) increasing accuracy (see also note 75).
A fixed-point example where a similar approach might be applied is:

```
long fract = long fract + fract * fract
```

The current specification requires the fract * fract part to be rounded to fract before the addition is done, thereby loosing precision and (often) requiring additional instructions. A similar case is

```
fract = fract1 * fract2 * fract3
```

where also intermediate rounding takes place. It should be investigated whether the floating-point `contracted expression' approach can be extended to include fixed-point arithmetic. If this is done, warnings similar to those in note 75 of the C standard should be repeated in the fixed-point context.

Response: Rejected. This functionality can already be achieved using casts; it is preferred to wait until more experience is gained; words to this effect will be added to the rationale.

NL-6 It is felt that the penultimate paragraph of 2.1.6.2.1, dealing with the special treatment of the values 1 and -1 as resultvalues of a multiplication operation, is too special and confusing. For instance: why also include the value -1 here which can always be represented? And what happens with this `saturating' behaviour when the state of the FX_OVERFLOW switch is MODWRAP? It is proposed to remove this special treatment.

Response: Rejected; however, the text will indicate that this special treatment is only specified to support specific implementations that otherwise would not be conformant, and will be deprecated in the future.

NL-7 The reasons to include unsigned fixed-point arithmetic in the document (orthogonality, analogy to integer arithmetic, some processors support it) are fully endorsed. Still, it is realized that the feature does add to the complexity of the specification (doubles the amount of types, requires special treatment etc). It might be useful to include more justification. Is there a striking example (hardware or application) that benefits from unsigned fixed-point arithmetic that can be described?

Response: An example (possibly on fuzzy logic) will be added.

NL-8 The need for and the usage of the FX_OVERFLOW state DEFAULT is not clearly described. Maybe DONTCARE is a better name for this state, as it is intended to be used for those pieces of code for which overflowhandling is not critical.

Response: Partially accepted: the fact that the DEFAULT state indicates that overflow behaviour is not critical will be better explained; the undefined behaviour for DEFAULT will be retained, because that will allow implementations to choose the most optimal code. The term DEFAULT is already used in the standard for this type of cases.

NL-9 The current text in section 3.2.1 allows address-space-qualifiers to be used for struct- and union-members; this should be disallowed.

Response: Accepted; constraint should be added if necessary.

NL-10 The current text in section 3.2.1 allows address-space-qualifiers to be used in function prototypes and function declarations; is that the intention, or should it be disallowed?

Response: Not allowed for now, to be revisited when we have prior art.

NL-11 Section 3.2.2 (Processor register access) introduces a concept of register variables with file scope. This is highly irregular and errorprone. It is proposed to either move the description of this functionality from the main body to the annex, or, preferably, to remove the functionality completely.

Response: Rejected; the text will be revised though.

NL-12 It should be an error when an address-space-qualifier is used in conjunction with a register declaration (section 3.2.5.1).

Response: See NL-11.

Comments from United Kingdom

Report from IST/5 UK National Body

UK-1 The material in this technical report addresses specialised problems in niche environments. To require support for these features from all compiler vendors would impose unwarranted hardship on members of the broader community. In the front matter a disclaimer should be added that there is no intention of incorporating this material into a future revision of the C standard. If incorporation were proposed today in the current form, the UK would vote NO.

Response: There is no intention to pre-empt the future question on what to do with the material in the TR when revision of the C standard is considered. It is recognized that the issue of the ever increasing size of the C standard is an issue that needs to be addressed by WG14.

There are technical defects in the proposed TR, but so long as they never become part of the C Standard the damage they can do should remain minimal. These are our other comments:

UK-2 p. 11, 2.1.5: The mechanism of using suffixes to denote the type of literals is not scalable. This affects not just fixed-point data types but also character literals and format specifiers. A better mechanism should be devised for some future revision of the C standard. This is an opportunity for the C and C++ committees to collaborate on a common solution to a common problem.

Response: The problem is recognized, cannot be fixed within the scope of the TR.

UK-3 p. 11, 2.1.5: Using 'q' as a suffix for _Accum literals conflicts with existing practice for designating "quad" floating point numbers on Itanium processors.

Response: Accepted; the new suffix will be the letter 'k'. The letter 'k' will also replace the letter 'q' in the names of the various functions, and as format specifier in scanf/printf related functions.

UK-4 p. 9, 2.1.3: Fixed-point overflow has undefined behaviour by default. Undefined behaviour is evil. The default overflow behaviour should be implementation-defined.

Response: Rejected: there is an analogy with signed integer overflow. This issue should be addressed separately with the more general problem.

UK-5 p. 32, 7.18.6.2: When rounding, "If the value of n is negative or larger than the number of fractional bits in the fixed-point type of f, the result is undefined." An "undefined result" is not a defined term; it should say the result is unspecified.

Response: Accepted; text will be changed.

UK-6 p.64, B.1.2: (#pragma addressmod) The grammar of the language does not allow preprocessor directives to extend over more than one logical line. Each physical line should end with a '\' character. The ',' following "Write_access" should be moved inside the square brackets (assuming it is optional). The trailing ';' is unconventional and doesn't appear to add any meaning.

Response: Accepted; text will be corrected.

UK-7 p. 42, 3.2.3.3: If memory-space-modified pointers are restricted to referencing that address space, what should happen if an alien address is assigned to them? Compiler

diagnostic? Seg fault? Undefined behaviour? The document should specify the behaviour in this case.

Response: Accepted; the behaviour will be undefined; see also US-23.

UK-8 p. 42, 3.2.4: If memory-space-modified pointers are restricted to referencing that address space, then a pointer to nested memory space A should NOT be used to point to objects in enclosing named address space B. Remove the "implementation dependent" possibility of doing so from the document.

Response: Accepted; however, the behaviour will be undefined; see also US-23.

UK-9 p. 42, 3.2.3.3: Is it necessary to use some kind of memory-space-cast to assign a value from an unmodified pointer (which can address any memory space) to one which is restricted to the named space? What syntax should this take? Note that 3.2.6 refers to an "implied cast to an unmodified pointer". Do such conversions need to be spelled out in connection with the Standard's paragraphs on casts, conversions, promotions, etc? Pointers to different address spaces are allowed to be different sizes (3.2.4). Can a memory-space-modified pointer be a function pointer?

Response: Accepted; the text will be improved (in response to US-23) to address these issues.

UK-10 p. 43, 3.2.5.1 s/that are do need to have/that do not need to have/

Response: Accepted; text will be corrected.

UK-11 p. 43, 3.2.5.3 "int myspace[10]; /* not allowed */"-- presumably should be something like: "int myspace arr[10];" ?

Response: Accepted; text will be corrected.

UK-12 p. 43, 3.2.5.3 ... But this (UK-11) raises the question, what is the scope of named address space identifiers? Can they be hidden by declarations in a nested scope, like other identifiers? This should be explained in the document.

Response: Accepted; the text will specify that the namespace in which the address space identifiers are defined is the global namespace, and that the address space identifiers behave as if they are predefined or typedef'd at the beginning of the program.

UK-13 p. 43, 3.2.6 "Code will then port between different target platforms." This sounds a bit vague (not to mention pie-in-the-sky). Source code or executable code? Does "different target platforms" refer to different compilers targeting the same processor or embedded system? Or does it mean different processor systems, which may happen to have the same kinds of memory space but arrange them differently? (Some embedded systems have RAM and ROM on a single chip, others use a combination of chips.) Clarify.

Response: Accepted; paragraph will be redrafted. The portability applies to systems with similar memory configurations; portability is at source code level (not at execution level).

UK-14 p.64, B.1.2, and p. 74, D.2: There seems to be some overlap between the address spaces proposal and the hardware i/o proposal, at least as relates to specifying the base address and the register width. If there is duplication or conflict, these should be harmonised.

Response: Accepted; text will be revised.

Comments from The United States

US-1 2.1.5 Fixed-point constants
The suffixes 'q' and 'Q' are already in use by several implementers for quad precision floating-point. Need to find a different suffix.
Response: Accepted; see UK-3.

US-2 2.1.6.2.1 Binary arithmetic operators
What is the result of fixed-point type operator floating-point type? Rationale implies it is floating-point type.
Response: Accepted; text will be modified to ensure that the result is float.

US-3 2.1.7 Fixed-point functions
The suffix 'q' is already in use by several implementers for quad precision floating-point. Need to find a different suffix.
Response: Accepted; see UK-3.

US-4 Section 6.4.4.3 Fixed-point constants
The suffixes 'q' and 'Q' are already in use by several implementers for quad precision floating-point. Need to find a different suffix.
Response: Accepted; see UK-3.

US-5 7.18 Fixed-point arithmetic. The suffix 'q' is already in use by several implementers for quad precision floating-point functions. Need to find a different suffix.
Response: Accepted; see UK-3.

US-6 7.18.6.3 The fixed-point countls functions
If the argument is zero, the result should be just N-1 (not at least N-1).
Response: Rejected; however the text will indicate that (exactly) N-1 is the recommended return value.

US-7 3.2.5.1 Register storage class 'that are do need' needs to be reworded.
Response: Accepted.

US-8 Circular buffers
'in various processor is so divers' needs to be reworded. Perhaps 'divers' should be 'diverse'?
Response: Accepted.

US-9 We are very uncomfortable with pragmas being used to define identifiers as proposed in Annex B for address space qualifiers and in Annex D for "access_spec"s. *BDTI expects it would feel obliged to vote against the final document if it contains such pragmas.*
Response: Accepted; the construct is not ideal. It will be investigated whether a better approach can be used.

US-10 A topic currently overlooked by the technical report is the need to define and create objects (usually arrays) with more restrictive alignments in memory than the default. For the best efficiency it would not be unusual, for example, to require that an array of 16-bit

"fract"s begin on a 16- or 32-byte boundary. Often these requirements arise for reasons unknown to the compiler, so the compiler cannot always be counted on to ensure the necessary alignments automatically. Two new features would address this concern: Variants of "malloc" and "realloc" that take an additional alignment parameter for objects allocated on the heap, and a new standard pragma for objects not allocated on the heap.

Response: The problems are recognized; the issue is however considered to be more general, and should be addressed at language level.

US-11 Section 2, Fixed-point arithmetic:
It would be more convenient if the "FX_OVERFLOW" pragma were split into two versions, one controlling "fract" overflows ("FX_FRACT_OVERFLOW"), and the other controlling "accum" overflows ("FX_ACCUM_OVERFLOW"). It is not unusual to need "fract" operations to saturate and at the same time to know that "accum" operations will never overflow. If turning on saturation costs overhead on a machine, it would be better to pay that overhead only for the "fract" operations that might overflow and not for the "accum" operations. Currently, making this distinction requires turning "FX_OVERFLOW" on and off between "fract" and "accum" operations, or, alternatively, forgoing the pragma and fastidiously using the "sat" keyword with "fract" operations but not "accum" ones. (The way "sat" is propagated through operations doesn't always make this completely trivial, either.)

Response: Accepted.

The committee should consider whether there aren't too many fixed-point types. Two options for reducing the number of types:

US-12 Eliminate the overflow qualifiers "sat" and "modwrap". They add complexity to the type system and would probably not be more convenient than the two pragmas "FX_FRACT_OVERFLOW" and "FX_ACCUM_OVERFLOW", assuming both were provided. (BDTI originally favored the overflow qualifiers, but that was before pragmas were part of the proposal.)

Response: Rejected. However, in response to the more general comment on the number of fixed point types it is decided to remove the 'modwrap' functionality from the document, and mention 'modwrap' only in a separate clause in Annex F (Functionality not included in this TR).

US-13 Eliminate all the unsigned fixed-point types except "unsigned short fract" and "unsigned short accum". As far as we know, no one has claimed a need for the larger unsigned types, other than a kind of "inertia of consistency".

Response: Rejected (see also US-12).

Among other benefits, a reduction in the number of types would dramatically cut the number of identifiers that must be defined by a C++-compatibility header (Annex G).

The following natural operations are not directly supported:

US-14 integer * fract -> integer

Response: Accepted; see NL-3.

US-15 integer / integer -> fract

Response: Accepted.

US-16 integer / fract -> integer

Response: Accepted; see NL-4.

US-17 fract / fract -> integer
Response: Accepted.

Functions or macros should be supplied for these operations (although probably not for all possible combinations of types).

US-18 The "fract" functions "abshr", "absr", "abslr", "roundhr", "roundr", "roundlr", "rounduhr", "roundur", and "roundulr" (2.1.7.1 and 2.1.7.2) should be defined to return a saturated result if the true result cannot be represented; otherwise, the functions won't be useful in many situations.
Response: Accepted.

US-19 A type-generic "fxbits" function (2.1.7.6) isn't possible, because the specific function to be substituted is not uniquely determined by the type of the operand. While the complement type-generic "bitsfx" function is feasible, neither "bitsfx" or "fxbits" is really needed, so the simplest course is to drop both.
Response: Accepted.

US-20 In Section 2.2 on specific changes:
The sentence *If an argument has fixed-point type, the behavior is undefined.* should not be added to Section 6.5.2.2. It should be possible to call non-prototyped functions with fixed-point arguments just as for other types.
Response: Accepted.

Section 3, Multiple address spaces support:
US-21 The keywords "const", "volatile", "restrict", and "register" should have their usual meanings when mixed with address space qualifiers (3.1.3 and 3.2.5.1).
Response: Accepted except for register; leave only first line of 3.1.3; remove 2nd sentence of 3.2.5.1.

US-22 Address space qualifiers for registers are problematic (3.2.2); for example, on most systems it won't be possible to have a pointer into any of these pseudo-address-spaces. Section 3.2.2 doesn't add anything to the technical report and should be dropped.
Response: Rejected; see also NL-11.

US-23 A cleaner, more general model is needed for the nesting of address spaces (3.2.4) and for pointers into address spaces (3.2.3.3 and elsewhere). We would suggest something along the lines of the following model (but not this specific text):
Every object exists in some address space (and possibly in multiple address spaces, if address spaces overlap). If the type of an object includes an address space qualifier, the object exists in the specified address space; otherwise, the object exists in the _generic_ address space. An implementation must support the generic address space (of course), and may support other, named address spaces. For any two address spaces, either the address spaces are disjoint, they are the same, or one is a subset of the other. Other forms of overlapping are not allowed. The implementation must define the relationship between all pairs of address spaces. (There is no requirement that all named address spaces be subsets of the generic address space.)

> As determined by its type, every pointer points into a specific address space, either the generic address space or a named address space. A pointer into an address space can only point to objects in that address space (including subset address spaces). A pointer into address space A can be cast to a pointer into address space B, but such a cast is undefined if the source pointer does not point to an object in B. (If A is a subset of B, the cast is always valid; if B is a subset of A, the cast is valid only if the source pointer points to an object in B.) A constraint requires that if a pointer into address space A is assigned to a pointer into address space B, then A must be a subset of B. (This constraint can be avoided with a cast.)

Response: Accepted.

Section 4, Basic I/O hardware addressing

US-24 The introduction to Section 4.3 says that

> An implementation is allowed to implement the interface by use of inline functions, intrinsic functions, or intrinsic function overloading, and still be conforming, as long as the interface seen from the user source remain the same.

But "iord" cannot be implemented as an inline function if it is really supposed to return different types depending on the register size, because then the function has no specific return type. A similar issue applies to the argument type of "iowr", etc., unless the inline function is defined to have an argument type larger than any possible register type. The C Standard does not define "intrinsic function" or "intrinsic function overloading", and we do not know for sure what these mean (or how they could be anything other than the same thing).

We continue to believe that function implementations of these operations should be allowed, but that implies at a minimum that different "iord" and "iordbuf" names must exist for different sizes: "iord" for "int" and smaller sizes, "iordl" for "long int", etc.

Response: Accepted; will be brought more inline with current C practice.

US-25 There is little purpose to standardizing the function-like macros "io_abs_init", "io_abs_release" (4.3.3), and "io_abs_remap" (4.3.4), for the following reasons:

- The circumstances in which these macros can be used and what they do (particularly "io_abs_remap") are all highly system-dependent.
- In portable device-driver code, uses of these macros could just as easily be replaced by calls to ad-hoc functions (e.g., "initUARTRegs", "remapUARTRegs"), with definitions for these functions provided in the local "iohw_ta.h" header as needed. Performance of these functions is rarely going to be a critical concern; however, if it is, they can be implemented as macros or inline functions in "iohw_ta.h".

Removing the "io_abs_" macros from the technical report would not preclude them from being standardized at a later time if sufficient prior art develops to make it worthwhile.

Response: Rejected; however, 4.3.23 will be cut down to simple definition + short description; explanation will be put in the annex.

Annex B, Embedded systems extended memory support

US-26 We question whether it is necessary to suggest a means of defining application-defined address spaces in this report (B.1.2). Since important details of the suggested pragma are left as implementation-defined anyway, there is little standardization value in presenting it. Implementations can already support such mechanisms as value-added extensions without requiring explicit sanction from the WG14 committee.

Response: Accepted (see also US-9).

Annex D, Generic "access_spec" descriptor for I/O hardware addressing

US-27 If the report is going to propose a "consistent and complete specification syntax for I/O registers and their access methods", it should not be this syntax. Besides our general objection to pragmas that define identifiers, we feel the system being proferred is too ad-hoc and could be better designed. If a better system cannot be substituted in place of the current contents, this annex should be tabled for further study.

Response: Accepted; new syntax will be specified.

US-28 Some implementation-specific characteristics are labelled in the document as being "implementation-defined" when they probably should not be.

Response: Accepted in principle; all occurrences will be investigated.

Section 2, Fixed-point arithmetic:

US-29a Minor issue: To clarify how the fixed-point features are intended to be used, the main body of the report should include an example of calculating a scaled dot-product of two "fract" vectors. Code for this could be:

```
fract a[N], b[N];
long accum acc = 0;
for ( int ix = 0; ix < N; ix++ ) {
        acc += (long accum) a[ix] * b[ix];
}
fract z = acc >> SCALE;
```

(This example is without any explicit rounding and with default overflow handling.)

Response: Accepted; the example will possibly be put in a new section 2.1.6.4.

US-29b Minor issue: Versions of the "round" functions for unsigned fixed-point appear to have been left out of Section 2.1.7.2.

Response: Accepted.

In Section 2.2 on specific changes:

US-30 Paragraph 3 of Section 6.7.2 should not be changed to read

> The type specifiers "_Fract", "_Accum", "_Complex" and "_Imaginary" shall not be used if the implementation does not provide those types.

In the context of the technical report, the implementation is already assumed to provide all the fixed-point types.

Response: Accepted.

US-31 For the same reason, Section 6.7.3 should not include the sentenceThe type qualifiers "_Sat" and "_Modwrap" shall not be used if the implementation does not provide those qualifiers.

Response: Accepted.

Section 4, Basic I/O hardware addressing

US-32 Minor issues: The "iohw_ta.h" header file used in examples is not a standard header file, and so the proper syntax is

```
#include "iohw_ta.h"
```

not

#include <iohw_ta.h>

This occurs in the annexes, too.

Response: Accepted.

US-33 Also, it is not obvious to us what the "ta" stands for.
Response: Accepted; change name to "iohw_my_hardware"

US-34 Section 4.2.6's exposition on "access_base_spec"s is not as clear as it needs to be.
Response: Covered by US-25.

US-35 Section 4.3.4 states: Use of "io_abs_remap" and "access_base_spec"s often provides a faster alternative than passing an "access_spec" as a function parameter.
Earlier the report calls "access_spec"s macros (4.2.5). We don't understand how a macro could be passed as a function parameter.
Response: Accepted; text will be modified.

Annex B, Embedded systems extended memory support
US-36 Minor issue: There is a Section B.1 but no B.2 or higher.
Response: Rejected; this numbering is conform the ISO Directives.

Special Note: it was decided that, after discussions with WG21, to reduce the description in Annex G on C++ compatibility to the bare minimum.