**Doc. No.:** ISO-IEC/JTC1/SC22/WG14/**N916**
**Author:** Martin J. O'Riordan
martino@theheart.ie (Convener to AGITS/WG9)
**Date:** 25 May 2000

# Irish Comments on JTC1-N6089 Extensions to C for Embedded Systems

| | |
|---|---|
| **Title:** | Programming Languages and Software Interfaces |
| **Scope:** | Programming languages, their environment and systems software interface, including specification techniques, common facilities, and language bindings. |

## Recommendation by NSAI/AGITS/WG9

The Irish panel recommends:

- **not** supporting the FXP and circular buffer extensions in their current form (WG14/N907) – the proposal lacks generality
- **not** supporting the inclusion of extensions to the pre-processor (WG14/N909) under any circumstances
- **not** supporting the hardware I/O proposal in its current form (WG14/N910) – the proposal lacks generality

Which translates to a NO recommended vote with comments.

## DISCUSSION AND COMMENTS

### *N907 - Fixed Point & Circular Buffers, etc.*

[NOTE: this paper was also presented to the C++ committee [WG21] in October 1999]

**Fixed Point**: The C++ programming language does not need language extensions to describe FXP data types. In C++, this would be better described using template classes, providing rich integration and extension. If such was added to the C++ standard, then it

would be possible for a conforming compiler to provide an optimal implementation for the data type without necessarily using the conventional implementation approach (i.e.: use intrinsic mechanisms instead).

Since this is a proposal for C and it is desirable that C++ will eventually inherit extensions to C, it makes poor sense to create a language extension for C, which would be better addressed by a library in C++. To this end, we believe that any FXP extensions to C should be formulated only in terms of a library-based interface. Such a library-based model would enhance the attractiveness of the proposal to C++, and once the names are standardised, boilerplate intrinsic code generation mechanisms may be used to map to optimal instructions.

The second problem concerns the generality of the FXP proposal itself. The extension is being proposed purely to facilitate binary FXP data types for use in embedded cryptographic and DSP applications. This lack of generality goes against the spirit of C. Decimal FXP data types are widely used among the financial IT community, and with the growing momentum of ***eCommerce***, it makes little sense to ignore this community when providing a standardised way of doing FXP. The development of *eCommerce* is very important to Ireland, which sees its involvement as a major player in this area in the future.

There is an identified need for binary and decimal FXP support in C. Our belief is that such an extension should handle the usual radii that C handles; namely: octal, decimal and hexadecimal. The generalisation to binary is also essential for embedded development, and it would be very desirable for binary radix generalisation to extend to all other multi-radix aspects of C (constants, etc.).

The OMG specification for Distributed Programming – CORBA – has a definition for Fixed Point that is constrained to Decimal, but which has far better flexibility and type safety. It would be worth pursuing a multi-radix definition for FXP modelled on the CORBA model.

**Memory Qualifiers:** The spectre of '`extern "..."`' in C++, and its ongoing semantic troubles is a potential problem. One of the issues with the C++ syntax is that it binds the word '`extern`' to things that are not necessarily '`extern`', for example:

```
extern "C" {
static void C_call_function ( int ) { ... }
}
```

where clearly the entity does not have external linkage. Worries about similar troublesome semantics being added to C do not bear thinking about. As N907 lacks a concrete proposal in this regard, this section should be deleted from the proposal.

**Circular Buffers:** This could be generalised to using integers with constrained range (syntax to be specified). Such integers would need also to specify their behaviour involving values that are outside of the specified range (especially increment and decrement). For example, instead of a pointer constrained to a circular buffer, a range restricted integer could be used:

```
int:0..41 index = 0;  /* values 0 through 41,
                         wraparound on overflow (i.e., MOD 42) */
int n;
char* pCircBuf = initialise_me;
/* scan the circular buffer three times */
for ( n = 0; n < (3 * 42); n++ )
    pCircBuf[ index++ ];
```

Compilers already routinely translate between indexed iteration of an array and pointer iteration of an array. They are generally normalised to the same IL. Recognising constant upper and lower boundaries for circularity optimisations is not a complex task, and if such boundaries corresponded to known optimal instructions for circular buffering, then it is quite simple. In fact, if the constants are to be always known at compile time (i.e.: no fat pointers), then recognition of circularity is trivial for the data flow analyser.

Using integers that are constrained to specified ranges makes it a lot simpler, as well as providing a more general mechanism. The integer in this case has a discrete type, and "circularity" is a side effect of ranges whose overflow behaviour is defined in terms of modulus arithmetic. There are three kinds of behaviour for overflow(underflow):

1. constrain the range to the upper(lower) bound of the range
2. wraparound within the range - circularity
3. raise an exception, '`signal`' or specify undefined behaviour

The discrete data type gives the compiler a lot of help in finding optimisations.

While it is not obvious how this functionality could be added to C without a language extension, in C++, the use of templates and template classes using standardised names would make it possible for a compiler to recognise special opportunity for optimisation using dedicated instructions on the target CPU, and thus avoiding the need for adding built-in language extensions.

## *N909 - Extensions to the Pre-Processor*

NO. There was no support for this paper among the members of AGITS/WG9.

There are already several excellent mechanisms available to achieve all of the facilities sought, such as Perl, Awk, Sed, M4, etc.; and that in the real world, developers neither require nor desire that a single, all-singing, all-dancing tool solves all of their problems. Such a tool would quickly become over-complicated and unusable. Real world development involves the use of many other tools, including 'make' and its derivatives, which are quite easily able to manage pre-compilation phases. The same argument that is being suggested for extension of the pre-processor could be used to add any extended functionality regarding source generation such as 'yacc' or 'lex' capabilities, even C++ (aka Cfront).

Also, these extensions are being proposed just so that implementation details of a given symbol can be hidden in a standard header. The C language says nothing about whether or not a standard include need be represented by a textual and syntactically valid file at all, so vendors already have the freedom to hide implementation tricks anyway they like.

## *N910 - Hardware I/O Access*

[NOTE: this paper was also presented to the C++ committee [WG21-N1233] in April 2000]

A standard way of specifying access to hardware may be desirable. However, despite the longevity of this proposal (circa 6 years), it seems to be poorly specified, immature and overly constrained.

Its biggest flaw concerns its presumption that all architectures are Von-Neumann, and that all busses are 8, 16, 32 or 64 bits wide. This is simply not the case. There are a lot of examples in the telecommunications and networking industry where Harvard or hybrid Harvard Architectures are used in the design of ASICs and FPGAs. These are often highly specialised processors and there may even be several processors on a single ASIC or FPGA, each with different bus specifications. Two real and current examples are:

- MAC Address Processor

  Application: Embedded Processor for Switch/Router

  Architecture: 48-bit data path
  16-bit code path
  16-bit data address
  10 bit code address

- Encryption Accelerator

  Application: Embedded processor for elliptic-curve encryption algorithms.

  Architecture: 256-bit data path
  32-bit code path
  24-bit data address
  12-bit code address

These architectures are simply not catered for at all by the Hardware I/O proposal in N910 (nor indeed are any form of vectorising or tagged architectures). An awful lot of modern embedded development is moving away from buying dedicated CPU chips and towards modular soft-specified CPU cores, with customer tailored specifications (see, CPort, Tensilica, ARC, ARM, MIPS, etc.). With these processors, the customer has considerable freedom in specifying the architecture. In all cases that were found, they are programming in C or assembler, and when written in C, they use their own libraries for hardware access. The limitations of this proposal would make it impossible to build a standard conforming C implementation for these configurations, and yet they represent actual, real world uses for embedded C development.

In Tokyo, Martin O'Riordan suggested that the specification should instead 'parameterise' the widths instead of providing a fixed set of pre-ordained values (the later examples illustrate this to an extent). This can be easily achieved using the existing C Pre-Processor, and a given implementation need only provide support for those 'specialisations' that are meaningful on the target architecture.

Again, the argument is that the proposal loses the general-purpose spirit of C. C as currently specified does not impose much by way of architectural constraint. The proposal in N910 goes completely against this historical spirit of C.

The other issue, which was identified in Tokyo, also concerns a lack of generality. In this case, concerning the “I/O with Transform” (OR, AND, XOR) functions. Here too is the presumption that only a fixed set of transforms exist. But there are several customisable processors that allow the instruction set to be altered or extended to incorporate special purpose functionality (e.g.: Tensilica, CPort). Uses often involve things like “SHIFT-MASK-

and-INVERT" field operations. Here too, the interface could be generalised to allow the concept of "I/O with Transform" to be parameterised. For example, the "transform" could be expressed as a parameter along the lines of:

```
#define io_trans(n,T,a,b) __io_trans##n(T(n),a,b)
...
#define IO_TYP(n)  uint_##n##t     /* or whatever */
#define IO_FCN(n)  IO_TYP(n)(*)(IO_TYP(n),IO_TYP(n))
```

with a set of predefined standard mandated constants (along the lines of SIGNAL) such as:

```
#define IO_AND(n)  ((IO_FCN(n))1)
#define IO_XOR(n)  ((IO_FCN(n))2)
#define IO_OR(n)   ((IO_FCN(n))3)
...
io_trans(8,IO_XOR, ... );
/* instead of the proposed */
ioxor8(...);
```

which the compiler could easily detect and optimise to the underlying best choice. At the same time a considerable reduction in namespace pollution results, as there is now one name '`io_trans`' describing an unrestricted set, instead of the 15 names for 15 forms in the proposal.

By approaching it in this way, it is also possible for user or vendor specific extended transforms to be added quite easily without interfering with the standard interface or prejudicing optimisations.

Its analogous implementation in C++ would be to use templates, but there is no reason why the concept of parameterisation could not be used in C too.

Finally, it has been observed that the "Memory Qualifiers" of N907 could be used to make all I/O hardware appear in the natural symbol space of C, which would certainly reduce the need for any library based interface to exist. For example, let's say that an '`extern "..."`' like extension '`storage "..."`' was added (not proposing that it should, but ...), then:

```
storage "ReadWrite I/O" {
volatile unsigned int blah;
}
```

coupled with greater specification for sequence points and volatile (N908) would be all that was necessary to interface in an architecture independent way, using the full C language to interface with arbitrary architecture independent address spaces.

```
blah ^= 5;
```

is a lot more meaningful than:

```
ioxor8 ( blah, 5 );
```

especially since it can not be incorporated into arbitrary expressions without writing everything as function calls. We are not proposing any syntax for stating the address space and access semantics of the names, but we do advocate that the semantics of its access could be closely coupled to its declaration. The declarative information is absolutely all the compiler needs to ensure that it does its code generation using the appropriate model.

The disadvantage of this approach however, is the implicit effect on the type system, especially once indirection is involved; and it is here that the brutal legacy of '`extern "..."`' from C++ would show it's ugly head.

The members of AGITS/WG9 were uncertain as to whether we should NOT support this proposed TR because of the deficiencies of this and other contributing papers; or whether we should support the idea of standardised hardware I/O in principle, but not in the form presented by this paper. We decided that to vote YES with comments was less likely to produce the desired results and concluded that to vote NO with comments was appropriate.

# IN SUMMARY:

**N907:** While FXP is important, it should not be constrained only to the requirements of the embedded arena of signal processing.

Circular buffers are not the only special purpose facility provided by embedded special purpose processors -- why add this and not other customisations! As a guiding principle, we should only add generalised mechanisms -- customisation is the natural domain of the vendor.

In conclusion, the proposal is supported in principle, but it needs more work by way of generalisation.

**N909:** No way. There are already appropriate tools available.

**N910:** While standardising hardware I/O may be desirable, this particular proposal suffers very badly from lack of generality, and we are moving towards the belief that it is attempting to solve the wrong problems. C does not distinguish between the concepts of memory mapped and non-memory mapped address spaces, nor do we believe that it should provide such a differentiation. It works purely with symbols and doesn't presume much about them at all.

A generalised addressing model derived from an extension of the storage specification of C (or linkage specification of C++) might be a better approach to allowing I/O to be managed natively by the whole language, rather than through a standardised library like interface.