Response on N916 - "Irish comments on N910 - Hardware I/O access."

Author: Jan Kristoffersen, Denmark, jkristof@ramtex.dk

Unfortunately the Irish comments in N916 contains a number of statements that are clear misunderstandings of the I/O hardware proposal N910 and not technically correct.

This response document is an attempt to clear this up and to summarize a few of the arguments / counter arguments, which have been brought up in the committee during the years.

N916 (page 4) states:
"Its (N910) biggest flaw concerns its presumption that all architectures are Von-Neumann,...."

This is the biggest misunderstanding in N916 of the N910 proposal. The standardization method proposed for I/O hardware access can be used with *all types of processor and bus architectures* on the market. In fact the most prominent task of the standardized I/O interface proposed in N910 is to encapsulate the characteristics of any underlying processor and bus architecture.

(This misunderstanding may come from the implementation examples generated in the C++ performance group. The purpose of these experiments was to test implementation methods based on C++ template in-line functions. To make these tests compile-able with the available C++ compilers only implementations for memory mapped I/O was generated.)

Background note:
Very early in the work on basic I/O hardware addressing it became clear that any standardization in this area could not be based on what existing processor architectures looked like. The diversity in processor and bus architectures turned out to be too high and the peculiarities too many.

By bringing peripheral I/O chips (I/O cells) in focus instead, the task became much simpler. The hardware interface to existing I/O chips on the market are reasonably well (defacto) standardized with relatively few variations. By strictly focusing only on I/O register (I/O chip, I/O cell) characteristics it became relatively easy to make I/O driver code portable across processor architectures at the source level, or to formulate this experience in a blunt way:

I/O registers are important. Processor architectures, bus architectures, and compiler implementations are not.

Later on this approach also seemed to be the obvious solution. After all, portability of I/O driver source code is the primary reason for doing this standardization. (It is our goal to standardize the syntax for I/O register access, not the characteristics of different types of processor hardware.)

N916 (page 4) continues:
"Its biggest flaw concerns its presumption ..... that busses are 8, 16, 32 or 64 bits wide."

No, the N910 proposal presumes that *I/O registers* (not *processor busses*) are 1, 8, 16, 32, or 64 bits wide (a subtle but important distinction).

The practical and historical reasons for this are:
- Most existing I/O chips on the market have a bus (and register) width of 8*2^n
- It is a continuation of the fixed sized types defined in *stdint.h*

Back ground notes:
There have been a number of arguments against this on the reflector during the years, both when discussing the *iohw* proposal and when discussing *stdint.h*.

As in N916 the, arguments have been that there exist processors on the market with, for instance, 12 bit or 24 bit busses and I/O registers.

Some counter argument (for both i*ohw* and *stdint.h*) have been that, yes there exist such architectures on the market, but A) code written for such special bit-widths will hardly be portable to other platforms, and B) there is nothing which prevents a compiler vendor from supporting such platform specific register (integer) widths along the same lines, *int12_t, int24_t* etc. For such cases, cross-platform portability will hardly be an issue anyway.

Furthermore, when people have been asked to come up with practical examples of I/O registers widths different from 8*2^n, instead of discussing some hypothetical cases, it turned out that in practice such odd sized registers were mostly (always?) located on the processor chip (i.e. on-chip I/O registers).
A situation which confirmed that the counter arguments above are valid in practice.

—

Finally people have argued that there exist situations where an I/O register only consists of (for example) 3 bits, where the rest of the bits are not implemented in the hardware. Should there be iord3(..) iowr3(..) functions?

Here a counter argument was that it is worth remembering some of the reasons for why the types in *stdint.h* were introduced:
- to be sure that the compiler integer type has enough bits to hold the (I/O register) value. (which cannot in a portable way be guaranteed with the standard types like *int* and *long*)
- to avoid using an integer type which will be grossly oversized (and inefficient) in some platforms just to be able to make the source code portable across multiple platforms.

The naming of the *iohw* functions *iord8, iord16* etc. originated from the same concept. I.e. more to make it possible to select an adequate access method for the register width rather than the wish to have the function name specify exactly how may bits are actively used by the application.

So if a 3 bit register is to be accessed then, for instance, the *iord8* syntax plus bit masking / shifting in the source code will be adequate as a standard method to assure good portability.

N916 (page 4) states:
"...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."

Unfortunately this is not correct. Historically the simple linear memory model used by the standard(s) has been the origin of most of the portability problems we see today with code operating on hardware. The constraints imposed by this simplistic memory model are too severe to provide a complete model for many of the existing processor architectures. The result is that intrinsic functionality is introduced (and required) by compiler implementations in order to handle hardware access with many existing processor architectures.

N910 proposes an interface for hardware access which is able to *encapsulate* such required intrinsic functionality in a standardized way, and the way *encapsulation* is done in the spirit of C is by the use of *functions*.

Background notes:
Many architectures have special addressing ranges which can only be accessed by use of special machine instructions (i.e. instructions which are not required during access of normal data memory). To access such addressing ranges from the C source level there must be some way to tell the compiler that special machine instructions for the hardware access should be generated. Therefore intrinsic functionality is needed. Unfortunately any attempts to standardize the required intrinsic functionality are doomed to fail in practice, simply because the processor and platform architectures are too diverse.

However, what *can* be standardized is an interface which are able to *encapsulate* any required intrinsic functionality in a standardized way.

N916 (page 5) states:
"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."

The *memory qualifiers* approach is prior art in embedded compiler design. However, historically a major drawback and limitation with this method has been that it is not possible for the user to add (encapsulate) more complex accessing schemes using the same syntax (for instance indirect access via base pointers, pseudo bus access, access via device drivers etc.). The reason is that operator overloading is not possible in C.

Memory qualifiers cannot *remove* the need for an API like interface, but they can usually make *implementation* of an interface much simpler.

Note that the memory qualifiers themselves usually prevent source code portability. Therefore the definition of the symbolic name for an I/O register (by use of memory qualifiers) must be isolated from the (portable) driver source code, for instance in a separate (non-portable) header file.

Background notes:
It may be useful here to summarize existing practice for I/O register access and emphasize what makes the approach in N910 special.

Prior art in compiler design has gone in two different directions: a) use of intrinsic *memory type qualifies* or b) use of intrinsic (inline) *functions*.

Example of a) use of intrinsic memory type qualifiers with existing compilers:

```
sfr P1=0xFFE0; /* (sfr = Special Function Register) define both the
                  access range, the address, and an implicit register
                  size in the given processor architecture */

#define P2 (*((unsigned char volatile _rom *) 0xe000 ))
           /* _rom specifies the access range, i.e.  that special
               instructions should be used for the access operations with
               the given processor architecture.  The register size is
               specified in the definition of the symbolic name */
unsigned char val;
val = P2; // The operation type is defined by a standard C operator.
P1 = val;
```

The advantages are :
- the symbolic port name definition is also a complete definition of the access method.
- the access operation is selected with standard C operators, so the source code is easily made platform independent.
- the code is easily optimized by the compiler.

The disadvantages are:
- a user can not extend the operator syntax with more complex accessing schemes.
- the syntax tempts programmers to think that I/O registers behave like traditional memory, which is very often not the case.

```
P1 = 0x23;// Fails if P1 is a read-only register
val = P1; // Fails if P1 is a write-only register
P1 += 2;  // Fails if P1 is not of type read-modify-write.
```

(Such subtle runtime failures has caused grief for many programmers. This, however, could be avoided if memory qualifiers were extended with access limitation qualifiers, so compiler diagnostics on I/O register access could take place.)

Example of b) use of intrinsic functions with existing compilers:

```
unsigned short _inpw( unsigned short port );
unsigned short _outpw( unsigned short port, unsigned short dataword );
/* The intrinsic functions (indirectly) define the size of the register, the
access range, and the operation type */

#define P1 0xFFE0 // The symbolic name only defines the address.
#define P2 0xE000

unsigned short val;
val = _inpw(P2);  // The operation type is defined by the function.
_outpw(P1, val);
```

The advantages are:
- A well-known and traditional API approach is taken.

- The very low-level approach makes address manipulations simple.
- Code can be optimized by macros or by making the API functions *inline*.
- A limited number of I/O functions reflects the nature of I/O hardware, where the I/O register itself or the processor hardware often only allow a limited set of operations.
- It is easy for the user to replace the API functions with API functions for I/O simulators etc. during development.

The disadvantages are:

- The functions themselves focus on the access mechanisms provided by the processor architecture which inevitably make any driver source code non-portable.
- The symbolic port name definition is too simple. As it is not a complete definition of the access method new function names have to be defined if new access methods are used on individual registers, which again makes the code less portable.
- If more than the few basic I/O register operations and access methods are supported then the number of API functions becomes considerable (namespace pollution).

The N910 proposal combines the idea of having a symbolic name which is a complete definition of the access method, like a), with the concept of doing encapsulation via standardized API interface functions, like b).

The advantages are:

- the symbolic port name definition is a complete definition of the access method.
- the access operation is selected with I/O register size specific (not processor architecture specific) API functions, so the source code become portable.
- Encapsulation is inherent. This enables a user to add new (more complex) accessing schemes, without any changes to the driver source code itself.
- Code can be optimized by implementing the I/O functions as C macros or by making the functions *inline*.
- Code is easily optimized by the compiler (when simple access mechanisms are used).
- It is easy for the user to replace the API functions with API functions for, for instance, I/O simulators during development.
- A reduced number of I/O operations (functions) reflects the nature of I/O hardware, where the I/O register itself often only enables a limited set of operations.
- The standard interface proposed by N910 can be implemented with any exiting C/C++ compiler on the market as long as it provides the required (intrinsic) functionality for I/O register access.

The disadvantage is:

- If more than the few basic operations are supported then the number of API functions become considerable (namespace pollution).

N916 (page 4) states:
"The other issue ... 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. ..."

This is only partly true. N910 reflects a pragmatic approach based on experience.

Some reasons for the reduced set of I/O register operations have been:
- To reduce the namespace pollution.
- To make implementations simpler by reducing the number of API functions. Reducing the number of standard operations (API functions) is especially desirable when a user is allowed to add support for new (more complex) access methods.
- Experience has shown that for simple access methods the compilers can usually generate fully optimized code anyway. For instance: `iowr8(P1,iord8(P1)+7);` may be optimized to a single machine instruction: `add P1,7`

So there is strong evidence that with respect to I/O access the advantage of supporting all arithmetic and logic operations in the language is relatively limited in practice.

Background note:
Some years ago there were discussions about entirely omitting OR, AND, XOR and only implementing the most basic I/O register operations, RD (read) and WR (write).

Some (historical) reasons for NOT doing this have been:
- Inspecting practical code has shown that OR, AND, XOR (bit set, bit clear, bit invert) are among the most common operations used on I/O registers (beside RD and WR).
- With more complex access methods more efficient implementations can usually be made for these common operations when they exist as separate functions.
- They are required with several (existing) compilers to spell out that special *and, or, xor* instructions for the access range should be generated, instead of : *rd-operation-wr* (i.e. as a way the programmer could handle poor optimizer's).

N916 (page 5) continues:
"... Here too, the interface could be generalised to allow the concept of "I/O with Transform" to be parameterised ..."

This is an interesting idea which deserves further exploration.

Some questions which should be answered are:
- is it a feasible way to simplify the interface for hardware I/O and reduce name space pollution, also if more complex accessing schemes must be supported ?
- to which extent does it require new compiler functionality?
- to which extent can it be implemented using existing compilers?

In summary

N916 contains a number of invalid statements which have mostly been based on wrong assumptions. Therefore, no substantial evidence has been provided that N910 should not be able to fulfill is purpose successfully as the proposal stands today.

However, N916 does contain some new ideas about methods to simplify the interface which might be worth considering.

An important property of the interface proposed in N910 is that it can be implemented using compilers of today. (In fact the standardization *method* proposed in N910 has been used in practice by a number of Danish companies for about ten years.)

In contrast the new ideas in N916 still need to be tested to see if they are feasible in practice. It would be natural if Ireland participated in such exploration within the frame of the current TR.

It is therefore recommended that Ireland change their vote to a YES (with comments).