Standardization of Basic I/O hardware addressing
in
ISO/IEC 9899:99 - Programming Language C.
Document N2767

Input from Danish MB to the ISO/IEC JTC1/SC22 - Plenary Meeting, August 1998, Copenhagen.

- To be discussed in conjuction with SC22/WG14 Work Program, “C” Standards

Basic I/O hardware addressing.

Denmark would like to see basic addressing of I/O hardware registers standardized in ISO/IEC 9899:99 - Programming Language C.

A need for standardization of prior art.

Today there is a lot of prior art for having compiler support of I/O hardware register addressing in C compilers for freestanding environments. It is also common practice that major parts of existing embedded application code depends heavily on efficient handling of I/O hardware register addressing from the C source level.

The major problems we see today comes from the fact that the syntax for doing basic addressing of I/O hardware registers were not standardized in ISO/IEC9899:89. As a result source code with I/O addressing operations written today will usually not be portable between compilers from different vendors, not even between compilers for the same processor architecture.

In order get current practice standardized, with C compiler support of basic I/O hardware addressing, Denmark strongly request that the syntax for basic I/O hardware addressing becomes standardized in ISO/IEC 9899:99 - Programming Language C.

Why standardize the syntax for basic I/O addressing

As embedded program applications grows in numbers and sizes it raises a growing need for reuse of program parts with new projects and for buying complete C function libraries for special application areas. Here the market for free-standing environments (embedded market) is far behind for instance the PC market. This is primarily caused by the large hardware diversity in the embedded market, and the simple fact that the most common I/O operations, basic I/O hardware addressing, does not have a standardized syntax in C.

A standardization of the syntax for the basic I/O addressing operations in C will give the individual programmer, the individual company, and the industry as a whole, considerable benefits:

- Easier programming. The same I/O syntax for all processors reduces the need for special knowledge about compilers and processors of a specific make.
- More flexible development cycle. For instance can an I/O driver be written and tested before the final selection of a specific processor type and compiler make.
- Source code will no longer be written to a specific compiler make but to standard C. This allow for instance standard PC compiler tools to be used for static test of embedded programs. An advantage in a project group when members compete for test time at the target system.
- Better possibility for reuse of I/O drivers. A well functioning I/O driver often represent a considerable know-how. If an I/O driver can directly be used by other programmers in the company, significant cost can be saved.
- Reduction in number of software versions. The same I/O driver source code can be reused across CPU families and C compilers from different vendors. This can give significant savings when code are developed for multiple processor

types. Typical application areas are communication protocols, drivers for serial communication etc.
- Software “components” for embedded use. A standardized I/O syntax will create a new market for embedded software libraries. The problem today is that the embedded market, because of C compiler dependencies, is too fragmented and therefore too small to be interesting for third party vendors.
- I/O driver support from I/O chip vendors. It becomes easier for chip vendors to deliver a general software support for their chips. Each software module will be able to cover a much larger market. In principle it become possible to reuse the same I/O driver module with all processor systems where the I/O chip itself can be connected.
- Generally applicable debugging tools and simulators. It becomes a lot easier to produce development tools based on C, which can be used across processor families and compilers makes.

The I/O addressing proposal

The I/O addressing method, as described in the current ISO/IEC JTC1/SC22 - WG14 document N731, propose a complete solution to the above standardization problems. The solution corresponds with a “standardization” practice which have been in use in the industry for many years.

The standardization method used is simply to define a few functions in the standard for doing basic addressing operations on fixed sized I/O hardware registers, plus an abstract *access_type* descriptor which represent a complete description of how a given I/O register should be addressed in a given hardware system.

This concept is new in the sense that it change the perception of I/O registers from something only related to processors busses and address ranges, to I/O registers as being individual object with individual properties, features, limitations and access mechanisms. This new concept enables and promote both I/O driver source code portability across multiple processor architectures and simplify the task of writing portable code.

Single register functions (prototype overview)
I/O functions for operations on single registers. For 8, 16, 32, 64 bit and 1 bit register sizes:

```
/* Read operations: */
uint_8t iord8(access_type);
uint_16t iord16(access_type);
uint_32t iord32(access_type);
uint_64t iord64(access_type);
bool iord1(access_type);

/* Write operations: */
void iowr8(access_type, uint_8t);
void iowr16(access_type, uint_16t);
void iowr32(access_type, uint_32t);
void iowr64(access_type, uint_64t);
void iowr1(access_type, bool);

/* AND operations (Clear group of bits) */
void ioand8(access_type, uint_8t);
void ioand16(access_type, uint_16t);
void ioand32(access_type, uint_32t);
void ioand64(access_type, uint_64t);
void ioand1(access_type, bool);

/* OR operations (Set group of bits) */
```

```
void ioor8(access_type, uint_8t);
void ioor16(access_type, uint_16t);
void ioor32(access_type, uint_32t);
void ioor64(access_type, uint_64t);
void ioor1(access_type, bool);
```

Buffer functions (prototype overview)
I/O functions for operations on I/O circuitry with internal buffers or multiple registers. Ex. a peripheral chip with a linear hardware buffer.
index is the offset in the buffer starting from the I/O location specified by *access_type*, where element 0 is the first element located at the address defined by access_type, and element n+1 is located at a higher physical address than element n.

```
/* Read operations on hardware buffers */
uint_8t iordbuf8(access_type, unsigned int index);
uint_16t iordbuf16(access_type, unsigned int index);
uint_32t iordbuf32(access_type, unsigned int index);
uint_64t iordbuf64(access_type, unsigned int index);

/* Write operations on hardware buffers */
void iowrbuf8(access_type, unsigned int index, uint_8t dat);
void iowrbuf16(access_type, unsigned int index, uint_16t dat);
void iowrbuf32(access_type, unsigned int index, uint_32t dat);
void iowrbuf64(access_type, unsigned int index, uint_64t dat);

/* AND operations on hardware buffers (Clear group of bits)*/
void ioandbuf8(access_type, unsigned int index, uint_8t dat);
void ioandbuf16(access_type, unsigned int index, uint_16t dat);
void ioandbuf32(access_type, unsigned int index, uint_32t dat);
void ioandbuf64(access_type, unsigned int index, uint_64t dat);

/* OR operations on hardware buffers (Set group of bits) */
void ioorbuf8(access_type, unsigned int index, uint_8t dat);
void ioorbuf16(access_type, unsigned int index, uint_16t dat);
void ioorbuf32(access_type, unsigned int index, uint_32t dat);
void ioorbuf64(access_type, unsigned int index, uint_64t dat);
```

Rationale for the I/O addressing proposal

The proposed solution allow standardized I/O addressing operations to be added to the existing draft for ISO/IEC JTC1/SC22 9899:99 Programming language C with no impact on the rest of the standard.

Function syntax versus an assignment syntax
The reasons for choosing a function syntax instead a assignment operator syntax are:

1. A function syntax allows standardized hardware register addressing to be added to the standard without the need to extend the existing native type system or the abstract machine concept used by the C standard.
2. A function syntax still allow compilers to implement I/O register addressing with exactly the same runtime efficiency as we have with current practice, but now using a standardized syntax. This can be done simply by implementing the new I/O functions as either function like macros or with the new keyword inline.
3. Functions is the C way to do encapsulation.
   Although the new I/O functions looks like API functions, the real purpose with the function syntax is to provide a portable and standardized mechanism for enabling use of the different compiler specific extended types needed for I/O access with different kinds of processor architecture, without sacrificing portability.
4. A function syntax provides an encapsulation of the underlying addressing method and provides a save and portable migration path for compiler support to still more advanced addressing mechanisms.
5. The standard I/O functions limits the types of operation done on I/O hardware registers to the native access operations (read, write) which can be accomplished on I/O registers with all embedded processor architectures, plus I/O functions for

the common bit-set and bit-clear operations (and, or) done on I/O registers. Other arithmetic operations is accomplished by use of these native access functions.
6. The function syntax emphasis the fact that I/O registers usually does not behave like memory cells in the sense that I/O registers often have an asymmetric read / write behavior. This again can simplify compiler implementations as it makes the use of ordinary arithmetic operations on asymmetric I/O registers a non-issue.

New abstract access type

The reason for using a new abstract access type are:

1. The new type provides a uniform and portable mechanism for referencing I/O registers in the C source code.
2. The abstract access_type provides a standardized method to isolate the actual addressing method used by a given I/O register with a given processor system from the C source code itself. In effect it enables a programmer to write C source code with hardware addressing operations in a compiler (and platform) independent manner.
3. It is a continuation of good programming practice, where I/O registers are identified by symbolic names instead of physical addresses, and where symbolic names for I/O registers are defined in separate header files, to order to make the source code itself more readable and portable.
4. The access definition encapsulation provides a uniform, safe and portable migration path for the implementation of still more advanced compiler supported access mechanisms. For instance from direct-addressing over indirect-addressing to addressing via different access drivers and the use of different mapping methods.
5. A standardized access_type encourage vendors to implements still more advanced compiler diagnostics for illegal I/O registers operations in future compilers (for instance AND operations on a write-only register).

An abstract data type used for maximum portability

A very wide range of different processor architectures are supported by different C compilers for free-standing environments. How access_type's should be defined, implemented and eventually initialized will therefore depend heavily on the characteristics of the underlying hardware system. With many architectures the access_type implementation will need to be system dependent in order to be efficient. Because of this, and because processor and bus architectures are assumed to evolve rapidly in the next decade, the definition of access_type is intentionally made loose. The standard promotes source code portability by defining the access_type only as an abstract type with certain properties.

This however, does not prevent the overall standardization goals to be fulfilled, and there is prior art for using such abstract types in the C standard. For instance the FILE type. Some general properties for FILE and streams are defined in the standard, but the C standard deliberately avoid to tell how the underlying file system as such should be implemented or initialized. A similar philosophy is used with access_type.

Basic I/O addressing in C, not as a separate API standard

It is inadequate to have basic I/O addressing standardized as an API standard used by third party vendors.

Basic I/O addressing concerns the very basic operations done by processor architectures and the *access_type* must be directly supported by the C compilers in

order to have both efficient support for I/O at the C source level, and the same efficient machine code generation we have today with the different non-standardized solutions. With many processor architectures special machine instructions are required to make I/O operations, and compilers for free-standing environments are required to support special internal types in order to be able to implement I/O access at C level.

Because of this, and because the purpose with the I/O proposal is to standardized existing practice with free-standing environments, basic I/O hardware addressing is required to be standardized as a part of the C standard.

Informative or Normative but optional

Today the primary target for this standardization of basic I/O hardware addressing is the market for free-standing environments, which is also the market that will get the largest economical benefit from such a standardization.
However, to allow compilers for hosted environments not to implement basic I/O hardware registers addressing it is proposed that basic I/O hardware addressing is standardized in ISO/IEC 9899:99 - Programming Language C as an informative annex or a normative but optional annex.

Proposal History and Conclusion

The principle of I/O access “standardization” has been in practical use by the industry since the early nineties and had since then been adopted by several companies.
The first I/O standardization proposal was presented to ISO/IEC JTC1/SC22 -WG14 in 1995 and had since be refined at several WG14 committee meetings to the form it has today. The final wordings for Basic I/O hardware addressing had since 1997 been ready for adoption in the standard.

However C support for basic I/O addressing had not yet been voted in. The resistance against having basic I/O addressing support in C had mainly come from people representing companies which main business area is related to hosted environments.
People with practical experience with I/O hardware drivers and free-standing environments have in general been in favor for the proposal, but have been outnumbered at committee meetings.

The C standard covers both the market for free-standing environments and the market for hosted-environments. The need and request for a standardization of basic I/O addressing is mainly related to free-standing environment, and this standardization can be done in in ISO/IEC 9899:99 - Programming Language C without causing any harm to the market for hosted environments.

Denmark therefore recommend that a broader view is taken by SC22, and recommend that SC22 support inclusion of basic I/O addressing as an annex to ISO/IEC 9899:99 - Programming Language C, for benefit of the electronic industry as a whole.