INTRODUCTION AND RATIONALE FOR BASIC I/O HARDWARE ADDRESSING

C++ performance group

X Basic addressing of I/O hardware registers

As the C language has matured over the years various extensions for doing basic I/O hardware register addressing have been added to address limitations and weaknesses of the language, and today almost all C compilers for free-standing environments and embedded systems support direct access to I/O hardware registers from the C source level; but these extensions have not been consistent across dialects. As a growing number of C++ compiler vendors now enter the same market place, the same I/O driver portability problem become apparent for C++.

The C++ committee should therefore take steps towards codifying common existing practice in the market place, in order to provide a single uniform syntax for basic I/O hardware register addressing.

Ideally it should be possible to compile C or C++ source code which operates directly on I/O hardware registers with different compiler implementations for different platforms and to get the same logical behavior during runtime. As a simple portability goal the driver source code for a given I/O hardware should be portable to all processor architectures where hardware itself can be connected.

The standardization considerations described in the following originate from solutions developed for the C market place, and a similar standardization effort take place in the C committee. The problem domain is the same for C and C++, and the standardization method proposed is applicable for both languages.

X.1 New perception of I/O registers simplifies the syntax standardization.

A standardization method must be able to fulfill three requirements at the same time:

- The standardized syntax must not prevent compilers from producing machine code which has absolutely no overhead compared to the code produced by the existing non-standardized solutions. This speed requirement is essential in order to get widespread acceptance from the market place.
- The I/O driver source code modules should be completely portable to any processor system (from 8 bit systems and up) without any modifications to the driver source code itself. I.e. the syntax should promote *I/O driver source code portability* across different execution environments.
- The syntax should provide an *encapsulation* of the underlying access mechanisms to allow different access methods, different processor architectures, and different bus systems to be used with the same I/O driver source code.
  I.e. the standardization method should separate the characteristics of the I/O register

itself from the characteristics of the underlying execution environment (processor architecture, bus system, addresses, alignment, endian etc.)

Several different attempts to make an international standardization of a general syntax for basic I/O operations over the years, have failed when it come to meet these very important requirements from especially the embedded market place and the market place for free-standing environments.

The major reason for this is two fold: 1) that I/O registers have usually been treated as "another type of memory", 2) that I/O registers access has been thought of as something related to processor busses and address ranges.

The I/O standardization method proposed overcome these limitations by treating I/O registers as individual objects with individual properties which are fixed and independent of both the compiler implementation and the surrounding processor system.

There is prior art for this solution. Nearly identical syntax standardization methods have, with some limitations, been in practical used since 1991 with existing C compilers (C89) for free-standing environments.

It is worth to notice that although the overall goal with standardizing basic I/O hardware addressing is to promote portability of library source code, then the major challenge is to get a standardized solution which does not reduce execution performance, especially with respect to speed and code size overhead.

X.2 Important Standardization Objectives

It is important to keep in mind that standardized I/O access does NOT means standardized hardware. The goal is to standardize the *syntax* for I/O operations, not the platform functionality.

An I/O register has a fixed size and endian, which are independent of how standard C types are implemented by different compiler vendors and independent of the access methods supported by different processors architectures and bus systems.

Most important is the fact that I/O registers usually do not behave like memory cells. I/O registers have special individual characteristics:

1.	write-only	(Uni-directional)
2.	read-only	(Uni-directional)
3.	read-once	(New data at each read)
4.	write-once	(Each write triggers a new event)
5.	read-write	(Bidirectional, but read != write)
6.	read-modify-write	(Memory like)

Individual bits in an I/O register may have individual characteristics. Only true read-modify-write registers behave like memory cells. The above list also shows that I/O registers should be treated similar to *volatile* data types as default.

As processor architectures and hardware platforms ARE different, a standardization must also provide a method to separate the description of the hardware differences and addressing methods from the source code. The standardization method should *encapsulate* descriptions of hardware differences, for instance in a separate header file.

The best way to encapsulate differences in allowed I/O access methods, and at the same time to create a uniform syntax for I/O access, is by use of a few standardized I/O *functions (or class member functions)*. This correspond to the way encapsulation is done in the spirit of C/C++. (The functions may be implemented as in-line functions or macros for speed optimization.)

Normally, arithmetic operations on I/O registers cannot be performed or have no logical meaning. Often read-modify-write operations on I/O registers are prohibited by the actual hardware. Operators like: +=, -=, *=, /=, >>=, <<=, ++, --, etc. are only meaningful where the I/O register and the bus architecture both allow read-modify-write operations. These natural access limitations make it obvious that the committee only need to define functions for the most basic operations on I/O registers (Basic *read* and *write* as a minimum). The programmer can build all other arithmetic and logical operations on top of these few basic I/O access operations.

With many existing processor architectures I/O register access often requires use of special machine instructions to operate on special I/O address ranges. Thus an extension of the type system is needed in order to access I/O registers from the C/C++ source level. By using a *function syntax* for standardized I/O access, all use of processor and platform specific I/O access types (implementation specific types) will be isolated to the implementation of these basic I/O functions and to the definition of the *access type* for a register object.
In this way the language can define a basic I/O hardware addressing syntax, which are portable to any processor system, without extending the type system defined by the C/C++ standard.

It is worth to notice that although a function syntax makes basic I/O hardware addressing look like traditional library functions (API functions), the underlying intention is mostly to get a portable way to extend the type system with compiler (processor and platform) specific access types.

X.3 Standardized syntax for I/O access.

All the considerations above are taking care of by the proposed standardization solution. It defines a number of functions which:

- Supports the most common fixed register sizes.
  - 8 bit, 16 bit, 32 bit, 64 bit or 1 bit (logical)
- Supports the most basic I/O register operations.
  - Read, Write,
  - Bit set (Or) in register, Bit clear (And) in register.
  - Single register objects, register array objects.
- Defines a new abstract type for I/O register referencing : *access_type*
- Provides an uniform encapsulation method for hardware and platform differences.

– Provides an uniform header file name. <iohw.h>

Example:

```
#include <iohw.h> // Encapsulates I/O register access definitions
unsigned char mybuf[10];
int i;
// ....
iowr8(MYPORT1, 0x8); // write single register
for (i = 0; i < 10; i++)
    mybuf[i] = iordbuf8(MYPORT2, i); // read register array
```

This I/O syntax standardization method creates a conceptual simple model for I/O registers:
Symbolic name for I/O port = I/O register object definition.

The programmer only sees the characteristics of the I/O register itself. Thus the underlying platform, bus architecture, and compiler implementation are don't care during driver programming. This underlying system hardware may later be exchanged without modifications to the I/O driver source code.

X.3.1 Prototype overview

.

Single register access

I/O access functions for operations on single register object. For 8, 16, 32, 64 bit and 1 bit register sizes:

```
/* Read operations: */
uint_8t iord8(access_type_8);
uint_16t iord16(access_type_16);
uint_32t iord32(access_type_32);
uint_64t iord64(access_type_64);
bool iord1(access_type_1);

/* Write operations: */
void iowr8(access_type_8, uint_8t);
void iowr16(access_type_16, uint_16t);
void iowr32(access_type_32, uint_32t);
void iowr64(access_type_64, uint_64t);
void iowr1(access_type_1, bool);

/* AND operations (Clear group of bits) */
void ioand8(access_type_8, uint_8t);
void ioand16(access_type_16, uint_16t);
void ioand32(access_type_32, uint_32t);
void ioand64(access_type_64, uint_64t);
void ioand1(access_type_1, bool);

/* OR operations (Set group of bits) */
void ioor8(access_type_8, uint_8t);
void ioor16(access_type_16, uint_16t);
void ioor32(access_type_32, uint_32t);
void ioor64(access_type_64, uint_64t);
void ioor1(access_type_1, bool);
```

Register array access

I/O functions for operations on I/O register array objects. This can be I/O circuitry with internal buffers or multiple registers. For instance a peripheral chip with a linear hardware buffer.

The *index* parameter is the offset in the buffer (or register array) 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_8, unsigned int index);
uint_16t iordbuf16(access_type_16, unsigned int index);
uint_32t iordbuf32(access_type_32, unsigned int index);
uint_64t iordbuf64(access_type_64, unsigned int index);

/* Write operations on hardware buffers */
void iowrbuf8(access_type_8, unsigned int index, uint_8t dat);
void iowrbuf16(access_type_16, unsigned int index, uint_16t dat);
void iowrbuf32(access_type_32, unsigned int index, uint_32t dat);
void iowrbuf64(access_type_64, unsigned int index, uint_64t dat);

/* AND operations on hardware buffers (Clear group of bits)*/
void ioandbuf8(access_type_8, unsigned int index, uint_8t dat);
void ioandbuf16(access_type_16, unsigned int index, uint_16t dat);
void ioandbuf32(access_type_32, unsigned int index, uint_32t dat);
void ioandbuf64(access_type_64, unsigned int index, uint_64t dat);

/* OR operations on hardware buffers (Set group of bits) */
void ioorbuf8(access_type_8, unsigned int index, uint_8t dat);
void ioorbuf16(access_type_16, unsigned int index, uint_16t dat);
void ioorbuf32(access_type_32, unsigned int index, uint_32t dat);
void ioorbuf64(access_type_64, unsigned int index, uint_64t dat);
```

The functions should be defined in *iohw.h*. Beside these definitions *iohw.h* also contain definitions for *access_types*.

X.4 The access_type parameter

The *access_type* parameter used in the I/O functions above represent or reference a complete description of how the given I/O hardware register should be addressed in the given hardware platform. It is an abstract type with a well-defined behavior.

The implementation of *access_types* will be processor and platform specific. Depending on how a compiler vendor choose to implement *access_types*, the definition of an I/O register object or may not require a memory instantiation. For maximum performance it could be a simple definition based on compiler specific address range and type qualifiers, thus no instantiation of an *access_type* object will be needed in data memory. There is prior art for this.

This use of an abstract type is similar to the philosophy behind the well-known FILE type. Some general properties for FILE and streams are defined in the standard; but the standard deliberately avoid to tell how the underlying file system should be implemented or initialized.

X.5 Fixed sized data types

The data parameter and return parameters used in the I/O functions above are integer data types with a fixed (or minimum) bit precision. These integer types are now a part of C99 (*stdint.h*).

I/O registers have a fixed size independent of how a compiler implement the standard integer types. Data values for use with I/O registers should therefore always have a fixed (or minimum) size which are independent of the compiler implementation.

The purpose with the fixed sized types is also to allow the programmer to decide which precision is needed by an application. A programmer can then do code optimization without the risk of running into the portability problems which exist with the old *int* and *long* types.
For instance, with smaller processor architectures it is often very "performance expensive", with respect to execution speed and code size, if the a program uses integer data types with a precision larger than needed by the given application. Fixed sized data types is therefore a performance issue not only related to I/O.

The committee should consider adopting C *stdint.h* (or at least part of it) in C++.

X.6 I/O initialization

When talking about initialization and I/O drivers it is important to make a clear distinction between I/O hardware (chip) related initialization and platform related initialization. Typically there is three types of initialization related to I/O:
1. I/O hardware (chip) initialization.
2. I/O selector initialization.
3. I/O access initialization.

Here only I/O access initialization (3) are relevant for basic I/O hardware addressing.

1. I/O hardware initialization is a natural part of a hardware driver and should always be considered as a part of the I/O driver application itself. This initialization is done using the standard functions for basic I/O hardware addressing. I/O hardware initialization is therefore not a topic for this standardization process.

2. I/O selector initialization is used when, for instance, the same I/O driver code should service multiple I/O hardware chips of the same type.

A solution with this proposal is to define multiple *access_type* objects, one for each of the hardware chips, and then having the *access_type* passed to the driver functions from a calling function.
I.e. *Instead of having the usual (platform dependent) I/O selector initialization, it now becomes a selection between standardized access_type objects*.

Note, this indicates that it is important that a standardization method does not prevent a compiler implementation from generating efficient code for *access_type* parameter passing. (This is an

area which typically create performance problems with implementations for C89 compilers).

Beside this performance issue I/O selector initialization is not relevant with respect to basic I/O hardware addressing.

3. I/O access initialization concerns initialization and definition of *access_type* objects. This process is implementation defined to a large extend (i.e. platform and processor architecture dependent).

With most freestanding environments and embedded systems the platform hardware is well defined so all *access_types* for I/O registers used by the program can be completely defined at compile time. For such platforms standardized I/O access initialization is not a standardization issue.

With larger processor systems I/O hardware are often allocated dynamically during runtime. Here the *access_type* information can only be partly defined at compile time. Some platform software dependent part of it must be initialized during runtime.

When designing the *access_type* object a compiler implementor should therefore make a clear distinction between *static information* and *dynamic information*. I.e. what can be defined and initialized at compile time and what must be initialized at runtime.
Depending on the implementation method and depending on whether the *access_type* objects should contain dynamic information, the *access_type* object may or may not require an instantiation in data memory. The more information are static the better execution performance can usually be achieved.

A few processor independent methods for I/O access initialization could be defined by the standard. With C++ overloaded constructors may be a solution.

X.7 Standardization efforts in related areas

IEEE 1596.5-1993 . IEEE Standard for Shared-Data Formats Optimized for Scalable Coherent Interface (SCl) Processors.
This IEEE standard define formats for interchanging integer, bit-field, and floating-point data between heterogeneous multiprocessors. The intent is to support efficient data transfers among heterogeneous workstations within a distributed computing environment. The standard defines a number of fixed sized data types, the bit/byte ordering etc.

The problem domain may seem similar to basic I/O hardware addressing. However, the IEEE standard addresses portability of data interfaces on the *binary level*, whereas this proposal addresses general I/O driver *source code* portability within a high-level language.

Uniform Driver Interface (UDI)
The UDI standard defines an execution environment for device drivers. The intent is to create a “hosting” environment for device drivers with a uniform interface to both the operating system

and the system hardware. The UDI standard defines a large number of functions for different services which should be provided by an UDI implementation.

The UDI standard originates from standardization requirements related to hosted environments, PC's and work stations. The function interfaces and services defined in the standard therefore also clearly reflect hardware features and software solutions typical for those platforms.

A small fraction of the UDI standard touches the same requirements for basic I/O addressing as this proposal. However, the UDI standard defines rather complex interfaces which obviously not have been designed to meet typical performance requirements from the market place for free-standing environments and embedded systems. Neither seems the UDI standard particular suitable for the broad range of different platform architectures which exists in this market. The processor architectures in this market typically have I/O and bus compositions which are very different from PC's and work stations.

The UDI standard therefore address a specific market place and is no real alternative to a syntactical simple and generally applicable standardization of basic I/O hardware addressing provided by the language.

X.8 Common market for C and C++

It will be beneficial for especially the market place for free-standing environments and embedded systems, if source code for I/O hardware drivers could be written so the driver code can be compiled with both C and C++ compilers. In this market place C compilers are still dominant. With a "C like" syntax for basic I/O hardware access users could benefit from broader range source library products from 3. party vendors. A common syntax will also assure a more smooth transition in this market from C to C++.

Note that only the standardized function interfaces are required to be "C like". An implementation for C++ can still take advantage of templates, classes and other advanced C++ features .

By : **Jan Kristoffersen**,
RAMTEX International ApS
Box 84, Skodsborgvej 346, DK-2850 Nærum, Denmark
Phone: +45 4550 5357, Fax: +45 4550 5390
Email: jkristof@ramtex.dk Email (C/C++) jkristof@pip.dknet.dk