Document Number: **WG14 N871/X3J11**

RATIONALE FOR BASIC I/O HARDWARE ADDRESSING. (Draft wordings)

Title: RATIONALE FOR BASIC I/O HARDWARE ADDRESSING.
Author: Jan Kristoffersen
Author Affiliation: RAMTEX International AS
Postal Address: Box 84, 2850 Naerum, Denmark
E-mail Address: jkristof@pip.dknet.dk
Telephone Number: +45 45505357
Fax Number: +45 45505390
Sponsor: DS
Date: January 28, 1997
Proposal Category:
  __ Editorial change/non-normative contribution
  __ Correction
  __ New feature
  __ Addition to obsolescent feature list
  __ Addition to Future Directions
  _X Other (please specify) Wordings for rationale
Area of Standard Affected:
  __ Environment
  __ Language
  __ Preprocessor
  __ Library
    __ Macro/typedef/tag name
    __ Function
    __ Header
  XX Other (please specify) Rationale

N871 RATIONALE FOR BASIC I/O HARDWARE ADDRESSING
(Draft wordings)

X Basic addressing of I/O hardware registers

As the 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.

The C9X committee have had lengthy debates regarding codifying common existing practice in order to provide a single uniform syntax for basic I/O hardware register addressing.

Ideally it should be possible to compile C source code which operates directly on I/O hardware registers with different compiler implementations for different platforms and to get the same logical behaviour 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.

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

A standardisation method must be able to fulfil three requirements at the same time:

- The standardised syntax must not prevent compilers from producing machine code which has absolutely no overhead compared to the code produced by the existing non-standardised 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 standardisation 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 standardisation 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 standardisation method proposed overcome these limitations by treating I/O registers as individual objects with individual properties that are fixed and independent of both the compiler implementation and the surrounding processor system.

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

X.2 Important Standardisation Objectives

It is important to keep in mind that standardised I/O access does NOT means standardised hardware. The goal is to standardise 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 (Bi-directional, 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 standardisation must also provide a method to separate the description of the hardware differences and addressing methods from the source code. The standardisation 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 C syntax for I/O access, is by use of a few standardised I/O *functions*. (Which may be implemented as simple macros or in-line functions for speed optimisation)
This is corresponding to the way encapsulation is done in the spirit of C.

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 source level. By using a *function syntax* for standardised 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 standard.

It is worth to notice that although the 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 Standardised syntax for I/O access.

All the considerations above are taking care of by the proposed standardisation method (working document N731). The proposed solution 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 a uniform encapsulation method for hardware and platform differences.
- Provides a uniform header file name. <iohw.h>

<u>Example:</u>

```
   void iowr8(access8 addr, uint8_t value);
   void iordbuf8(access8 addr, unsigned int index);
  ---
   #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 standardisation method creates a conceptual simple model for I/O registers (Symbolic name = 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 programming. This hardware may later be exchanged without modifications to the I/O driver source code.

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 behaviour.

The implementation of *access_type* will be processor and platform specific. Depending on how a compiler vendor chooses to implement *access_type*, the definition of an I/O register object may or may not require a memory instantiation. For maximum performance it could be a simple definition based on compiler specific address types 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 C standard; but the standard deliberately avoid telling how the underlying file system should be implemented or initialised.

X.5 Future actions

Although the committee recognise free-standing environments as an important market place for the C language, it has not been able to reach consensus for adding support for basic I/O hardware addressing to C9X. This addition to the rationale shall therefore be seen as the committee's good intention to address this topic in future revisions of the C standard.