Date: Thu, 14 Dec 1995 20:52:14 +0100

Subject: Standardization of basic I/O hardware adressing

Document Number: WG14 N509/X3J11 95-110

STANDARDIZATION OF BASIC I/O HARDWARE ADDRESSING.

Title: I/O sypport on Embedded Machines
Author: Keld Simonsen & Jan Kristoffersen
Author Affiliation: RAMTEX A/S
Postal Address: Box 84, 2850 Naerum, Denmark
E-mail Address: jkristof@pip.dknek.dk
Telephone Number: +45 45505357
Fax Number: +45 45505390
Sponsor: RAMTEX A/S
Date: 1995-12-11
Proposal Category:
__ Editorial change/non-normative contribution
__ Correction
XX New feature
__ Addition to obsolescent feature list
__ Addition to Future Directions
__ Other (please specify) removes unnecessary restriction
Area of Standard Affected:
__ Environment
XX Language
__ Preprocessor
XX Library
XX Macro/typedef/tag name
XX Function
XX Header
__ Other (please specify) _____
Prior Art: n/a
Target Audience: n/a

Related Documents (if any): none

Proposal Attached: Introduction paper

Abstract:
The purpose with this paper is to define clear goals for a standardization of the syntax for simple Input / Output operations on hardware.
The primaery goal is to make I/O driver functions portable between different processor platforms (and between C compilers from different vendors).

1.0 STANDARDIZATION OF BASIC I/O HARDWARE ADDRESSING.

Summary

The purpose with this paper is to define clear goals for an

1/5

ISO-C standardization of basic I/O operations on hardware, and to present the framework for the proposal.

1.1 Standardization of interrupt operations in ISO-C.

The syntax for interrupt functions and for the interrupt-enable and interrupt-disable operations also need standardization. Standardization of interrupt operations in ISO-C is also an important issue, but it is a separate problem which should be handled separately. Standardization of interrupt operations is not a part of this proposal.

1.2 Definition of words

I/O register :	A register in a peripheral hardware device. "I/O register" is used both when the peripheral hardware device is physically located on the CPU chip (on-chip or internal I/O) and when the peripheral device is located on a separate chip (external I/O).
I/O :	Synonymous with I/O register
I/O device :	Synonymous with peripheral device
Peripheral device :	Hardware circuit which contains one or more I/O registers which can be addressed from a C program. A peripheral device can be looked upon as a collection of I/O registers.
Register :	Synonymous with I/O register in this paper.

2.0 PRIMARY GOALS

The primary goals with an ISO-C standardization of basic I/O operations on hardware are as follows:

*** GOAL A:
To be able to write driver functions which operate on hardware I/O registers, in such a way that the source code becomes independent of the processor architecture and a specific c compiler.

*** GOAL B:
To promote flexibility without sacrificing efficiency.
A solution must allow the compiler to generate in-line code in order to maintain efficiency with respect to execution speed.

*** GOAL C:
To be able to compile driver code for peripheral devices for syntax checking, before the processor platform is selected.

All suggestions for an implementation should be evaluated against these few primary goals.

2.1 Comments on the primary goals

2.1.1 The primary users

A standardization of simple I/O operations on hardware would be a tremendous benefit for all parties in the embedded processor market.
The large amount of different processor architectures, the situation where new processor derivatives come on the market every month, and the growing number of C-compilers from different vendors to different processors, creates a still growing need for a standardization.

As the primary "users" of the ISO-C standard in the future could very well be found within the embedded processor market, and as this market has very different processor architectures, a solution should take its standing point in the needs from this market.
Any solution, which can solve the problems for the embedded processor market, will most certainly also be versatile enough to be used on other markets.

2.1.2 Portability and reuse of driver C source code

It is a fact that the functionality and complexity of peripheral hardware circuity is constantly growing, and we see the driver source code needed, in order to use and control this hardware, also becomes still more complex. A well functioning driver code often represents a know-how which goes far beyond the source code itself.

Therefore, the goal is that the driver source code can follow the peripheral hardware circuitry, so it can be reused with different processor platforms and compilers, without the need for any modifications in the C source code itself.

2.1.3 Trend: standardization of peripheral I/O circuitry

The processor chip vendors now try to design peripheral I/O cell libraries for on-chip I/O, which can be reused when new processor chips are designed.

These years processor vendors work for a standardization of the on-chip interface connections to peripheral I/O cells, so the I/O cells can be exchanged between processors without the need for a redesign.

Examples of this are the IMB bus from Motorola, where the same I/O cells now are used in several processor families, and the "Peripheral Interconnect Bus" (PI bus), which is developed by the companies: Philips Semiconductors, Siemens, SGS Thomson, Advanced RISC Machines (ARM), and Termic/Martra MHS. The PI bus development project takes place within the frames of an EU project called "Open Microprocessor Initiative" (OMI).

A standardization of simple I/O operations in C will make it possible to reuse the software driver functions for I/O cells directly.
It will become possible to make portable software "libraries", which can be used with both processor on-chip I/O and individual peripheral I/O chips.

This portability of I/O drivers between processor platforms should be achieved without the need for any modifications to the I/O driver source code itself. Only the description of how the I/O circuit is physically connected should be updated.

2.1.4 What I/O registers have in common

The main purpose with the proposal is to standardize basic I/O hardware addressing, and therefore bring into focus WHAT I/O REGISTERS HAVE IN COMMON.

The intention with the proposal is not to handle all aspects with different processor architectures.
For example, some processors have special processor registers which can only be addressed by special instructions. The handling of such special processor features is not a part of the proposal.

****
There will always be special cases. This should not prevent the syntax for basic I/O hardware addressing from being standardized.
****

2.1.5 The hardware description

It is a fact that processor architectures and hardware platforms ARE different. These differences must obviously be expressed somewhere, before the compiler/locator can generate the right code.

****
The goal with the hardware description is to standardize the syntax which describes the connection between I/O circuitry and a processor kernel.
****

The hardware description for an I/O register must be a complete description of the access method which must be used by the compiler / linker / processor in order to get physical access to the I/O register.

It should be possible to isolate all changes, which have to be made when reusing an I/O driver function with another processor or hardware platform, to the hardware description.

Preferably, it should be possible to isolate the hardware description in a single file (ex. an include file), which then will be the only file which has to be updated in order to move the I/O driver functions to another platform.

2.1.6 Speed performance

As exactly the I/O operations are the raison d'etre for embedded processors, then I/O speed performance must not be sacrificed for a standardization of the I/O syntax.

Any solution must allow the compiler to generate in-line code

in order to maintain code efficiency with respect to I/O execution speed.

2.1.7 Static versus dynamic addressing

Generation of in-line code usually implies that the compiler processes the hardware description at compile time and then generate optimized machine instructions for static I/O register addressing.

Some embedded processors even have busses (I/O addressing modes) which only accept static I/O addressing. In such cases in-line code generation is a must.

Typically, code for embedded processor system uses static I/O register addressing most of the time anyway.
That is, one set of driver functions for one peripheral device.
Therefore, driver code flexibility is best achieved at source level, i.e. through an uniform syntax for I/O operations and an uniform hardware description.

Dynamic I/O addressing is only needed with more special applications. For instance if the processor system uses several units of the same peripheral device and a driver function could service all units. This implies that the I/O access description can be passed as a single parameter.

Dynamic I/O addressing implies that the hardware description for a given I/O register can used as a variable. For instance as a parameter in the function header:

```
void driver_function( hardware_description, char data );
```

3.0 SECONDARY GOALS

The secondary goals with an ISO-C standardization of basic I/O operations on hardware are as follows:

*** GOAL D:
To describe differences in I/O register access methods and hardware platforms with a uniform syntax, and in such a way that goal a is achieved.

*** GOAL E:
To be able to make extended type checking on I/O registers, in such a way that the compiler can detect if a given simple I/O operation is allowed on a given I/O register.
Extended type checking should detect errors like read operations on a write-only register.

*** GOAL F:
To be able to describe allowed access methods on I/O registers with an uniform syntax, and in such a way that goal e is achieved.

*** GOAL G:
To be able to substitute individual I/O register access operations in the driver code with software stubs.
For example for execution test of the application before the

peripheral hardware is present and for controlled simulation of peripheral hardware devices.

*** GOAL H:
Keep it simple.
That is, a solution should be simple to use by the programmer, not necessarily simple to implement by the compiler vendor.

Where fulfillment of the primary goals are mandatory for an I/O standardization, then the secondary goals set the guidelines for finding "the best solution" during the standardization process.

3.1 Comments on the secondary goals

3.1.1 I/O registers are NOT memory

I/O registers are NOT memory. I/O registers usually does not behave like memory and should NOT be treated like memory.

I/O registers have special individual characteristics:
- I/O registers can be uni-directional (read-only or write-only).
- An I/O register can return a separate value each time it is read (read-once).
- An I/O register can initiate a new hardware event each time it is written (write-once).
- An I/O register can be bi-directional, but the read value has no relation to a previous write value. (i.e. a read-only register and a write-only register on the same physical address).
- An I/O register can behave like a memory cell. (read-modify-write operations allowed).

Individual bits in an I/O register can have individual properties. Each bit can individually have any of the above characteristics.

I/O registers are special and should be treated in their own way.

3.1.2 Limit arithmetic operators for I/O registers

Standard C assumes that storage for all data types are memory and that all arithmetic operations can be performed on the data. This is not the case with I/O registers.

Arithmetic operations on I/O registers usually cannot be performed or have no logical meaning. Often read-modify-write operations on an I/O register is prohibited by the actual hardware.

C operators which involve read-modify-write operations on the hardware either cannot or should not be used with I/O registers.
Operators like: +=, -=, *=, /=, %=, >>=, <<=, ++, --, etc. are not meaningful for most I/O registers.

***

50

This proposal suggest that arithmetic operators are not allowed on I/O registers.
***

Such a limitation respects the nature of I/O registers without sacrificing performance and user flexibility.
(This limitation will possibly also simplify compiler implementations considerably).

In cases where arithmetic operators on an I/O register are meaningful the operation can equally as well be performed as 2 separate addressing operations, a read operation and a write operation.
ex: where += is implemented as separate operations: read I/O, ADD data, write I/O.

Note, this should not prevent a compiler from generating optimized code for the example above and use a single ADD machine instructions instead. For example in the cases where we both have an I/O register which allows read-modify-write operations and a processor which have an ADD instruction for the address bus where the I/O register is connected.

3.1.3 The access_type

***
I/O registers have special characteristics and should therefore be treated with a new special type: The access_type.
***

The access_type definition should give a full description of how the I/O register is connected in the given hardware.

The access_type definition should describe: How the compiler should make access to the I/O register (memory bus, I/O bus, psychical address etc). Which limitations on access operations the I/O register has in the given hardware (read-only, write-only etc.), and the data size or data width of the I/O register.

The access_type definition should be considered as a part of the I/O hardware description.

First of all, the access_type definition for an I/O register must contain all the information needed by the compiler/linker in order to generate the right machine code for addressing the given I/O register in the given hardware.

Secondly, an access_type definition enables extended type checking on I/O. Warnings can be issued at compile time if the program makes illegal operations on an I/O register, for example read operations on a write-only register.

3.1.4 I/O data sizes

In a given hardware system the actual I/O peripheral hardware and their connection to the processor hardware usually puts sever limits on which data operations can be used in practice on the I/O registers.

Usually an I/O register have specific data size preferences and must be addressed with machine instructions for that data size.

For example the I/O register may require that the processor uses:
- byte data addressing operations.
- word data addressing operations.
- long data addressing operations.
- bit data addressing operations.

****
Therefore, the hardware description for an I/O register should also define the data type operation which is required by the I/O register.
****
Definition of the I/O data size for an I/O register will allow the compiler to detect illegal data operations on a register at compile time. Ex word operations on a byte register.
****

As an I/O register will only work with processors which support the correct data addressing operations, the I/O driver source code will in practice only be used in places where it can be ported. Specification of the I/O data size for a given I/O register allows portability problems to be detected at compile time.

Example 1:
A peripheral device has an input register which can only be read as byte (char) data. Bit and word operations are not be possible with the peripheral hardware.
In this case a description of data size limitations will assure portability between processor platforms.

Example 2:
A peripheral device with an output register requires the use of bit write operations in order to operate properly. The driver source code contains bit I/O operations.
In this case processor addressing capabilities puts limits on the places where the peripheral device can be used, as not all processor architectures support bit operations on I/O hardware.

3.1.5 Syntax checking and software stubs

A standardized syntax, which allow driver source code with I/O operation to be compiled for syntax checking before the processor type is selected, would be beneficial.

During development of embedded software applications the still growing demands for a reduced time-to-market often put software development department in the situation where the software must be developed before the actual embedded hardware exists.

Another problem with testing I/O drivers for embedded applications are to get repeatable data from the real I/O hardware during debugging.
The reason for this is simply that it is difficult to control the state of the physical world outside the machine, which influences data from sensors etc. For instance is it a common

problem to test how well the embedded program reacts on various error conditions.

A standardized syntax, which allow I/O operations in a driver source code to be easily substituted with software stubs for the individual I/O registers, would be beneficial.
This will allow controlled simulation of peripheral hardware devices during debugging and promote better software testing.

4.0 THE FRAMEWORK FOR A STANDARDIZATION

In order to achieve a break-through in the standardization of basic I/O operations on hardware a new perspective is needed.

4.1 The I/O layer model

In the following the I/O layer model is presented.

The I/O layer model is a new mental model for I/O handling, which helps clarifying the standardization goals, and helps seeing different standardization problems in their right perspective.

The I/O layer model is a 2 or 3 layer abstraction model of the software and hardware. It separates I/O driver functions and I/O registers from the processor architecture and the compiler / linker.

```
I/O LAYER
  I/O driver function  <-->  I/O registers in a peripheral
                             device
  ------------------------------------------------------------
I/O DESCRIPTION LAYER
  I/O hardware         <-->  data bus / address bus connection
  description
  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
IMPLEMENTATION LAYER
  C compiler / Linker  <-->  processor architecture,
                             memory, stack
```

4.1.1 New mental model for I/O handling

I/O registers should be looked upon as a part of the (driver) application instead of as a part of the processor hardware.

Processor CPU, processor busses, memory and stack can be looked upon as just something which the program needs in order to be able to execute. These hardware parts belong on a lower abstraction level than I/O registers.

The standardized syntax for I/O operations in the source code creates a virtual link or gateway between the driver source code and the individual I/O registers in the peripheral device.

The compiler, the machine code, the processor and the bus architecture creates the physical link between I/O operations in the driver source code and the I/O register.

The I/O hardware description describes how the physical link between the source code and the I/O register should be made in the given hardware.

The standardized interface between the I/O LAYER and the lower layers makes the I/O driver functions independent of the hardware platform and the compiler.

If the I/O hardware description is also standardized and the hardware platform is the same then it should be possible to recompile the program with different compilers from different vendors.

4.2 Fulfillment of standardization goals

The I/O layer model makes it possible to set up simple guidelines for how a standardization of basic I/O operations on hardware can be done.

The interface between the I/O layer and the lower layers is the C syntax I/O operations. All hardware platform difference is isolated below the I/O layer. The link to the I/O register is described by a single access_type parameter.
Fulfillment of the primary goals A and C then only requires that the uniform syntax for basic I/O operations is selected.

Goal B, C and G indicates that hardware I/O operations should be looked upon both as in-line code (for efficiency) and as functions (for flexibility, type checking, easy prototyping and test).
This functionality can be implemented today in C with the substitution of function calls with macros, and in C++ with the in-line keyword and templates.

A simple I/O data type checking can be achieved today with I/O functions for each data type:

```
unsigned char iordby(access_type);
unsigned int iordwo(access_type);
unsigned long iordlo(access_type);
BOOL iordbit(access_type);
void iowrby(access_type, unsigned char);
void iowrwo(access_type, unsigned int);
void iowrlo(access_type, unsigned long);
void iowrbit(access_type, BOOL);
```

If function overloading is implemented in the new C standard then only 2 virtual functions are needed for I/O operations on hardware:

```
iord(access_type);
iowr(access_type, data_type);
```

This makes I/O operations simple to use by the programmer and fulfills goals A,B,C,G and H.

Fulfillment of goals D,E,F requires a closer look on the access_type parameter.

4.2.1 Standardized hardware description

The whole trick with the above I/O functions is to isolate the hardware description of an I/O register in a single parameter, the access_type;

By using the access_type parameter then I/O standardization of basic I/O operations on hardware simply becomes an extension of what today is considered normal good design practice.

Today it is common practice to let a symbolic name represent the I/O register and then in an include file define the symbolic name as being equal to the absolute physical address. With the access_type parameter the hardware definition is just extended a bit.

The extended I/O access_type definition should contain the following items:
1. Symbolic name for the I/O register.
2. I/O register data size (bit, byte, word, long).
3. Read / write limitations (for extended type checking).
4. Bus connection description (I/O mapped, memory mapped etc.).
5. The physical address.

Item 1,2,3 can fulfill the standardization goals A,C,D,E,F and item 2,4,5 allows the compiler to generate the right code for the given hardware platform.

4.2.2 Comments on the access_type parameter

When the symbolic name is used as the access_type parameter in I/O functions the compiler has all the information needed for making static I/O register addressing. The following I/O statements can all be compiled as speed efficient in-line code:

```
io_char = iord( PORT_A );
iowr( PORT_B, io_char );
iowr( PORT_B, iord( PORT_A ) + 0x10 );
```

With static I/O addressing extended type checking can be made at compile time.

With dynamic I/O addressing the access_type parameter can be implemented as a reference to a (global) access_type description:

```
void nibble_wr( access_type& ioreg, char value)
   {
   iowr( ioreg, value & 0xf );          // Low nibble
   iowr( ioreg, (value >> 4) & 0xf );   // High nibble
   }
```

In this example the compiler gets addressing information from the referenced access_type description.
The compiler must use the bus connection description and the physical address from the access_type description in order to select the right addressing mode.

4.2.3 Runtime typechecking on I/O operations

With dynamic I/O addressing extended type checking can only be made during runtime.

This raises the question: How can compiler detected runtime errors be handled in a standardized way? If exception handling like C++ is implemented in the new C standard a solution could be that the program throws an exception in case of an error. However, dynamic I/O addressing and dynamic type checking on I/O will probable find little use with embedded applications, as the hardware tend to be rather static.
We consider runtime typechecking be of minor importance compared to a standardization of the syntax for I/O operation.

4.2.4 Mental models for the access_type implementation

During implementation of the access_type parameter it can be looked upon as having a functionality which is a mixture of:
- a reference, like & in C++,
- a definition of a symbolic name with a compiler aided value assignment, like enum,
- as a definition of the elements in an I/O structure.

The whole I/O hardware description for a platform can be looked upon as an array of access_type definitions, where no code is generated for an element, if only static I/O addressing is used, and constant data is generated for an access_type element, so it can be referenced, if dynamic I/O addressing is used.

4.3 Forward and backward compatibility

The main purpose with the proposal for a standardization of basic I/O operations on hardware, is to incorporate a solution for both present and future needs in the upcoming new C standard.

The approach with using functions for I/O instead of direct I/O assignment correspond to the ideas behind the C++ classes, with implementation flexibility, encapsulation and information hiding.

The use of functions for I/O operations also assures backward compatibility.
If type-checking, dynamic assignment and function overloading is skipped, then this proposal for a standardized I/O LAYER syntax can be implemented directly with most of the C compilers for embedded processors on the market today. And usually without generating any overhead of machine code.

This ends the presentation of the framework behind the proposal for a standardization of basic I/O operations on hardware. Any comments regarding the topics in this paper would be highly appreciated.

Jan Kristoffersen. RAMTEX A/S
Fax: +45 45 50 53 90
E-mail: jkristof@pip.dknet.dk

57