Document Number: WG14 N656/X3J11

STANDARDIZATION OF BASIC I/O HARDWARE ADDRESSING.

Title: STANDARDIZATION OF BASIC I/O HARDWARE ADDRESSING.
Author: Jan Kristoffersen
Author Affiliation: RAMTEX A/S
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: 1997-January-31
Proposal Category:
__ Editorial change/non-normative contribution
__ Correction
XX New feature
__ Addition to obsolescent feature list
__ Addition to Future Directions
__ Other (please specify)
Area of Standard Affected:
__ Environment
__ Language
__ Preprocessor
XX Library
XX Macro/typedef/tag name
XX Function
XX Header
__ Other (please specify) _____
Prior Art: Widely used
Target Audience: Device driver programmers, I/O chip manufactures
Related Documents (if any):
WG14 N509/X3J11 95-110
STANDARDIZATION OF BASIC I/O HARDWARE ADDRESSING.
Introduction paper

WG14 N558/X3J11 95-110
STANDARDIZATION OF BASIC I/O HARDWARE ADDRESSING.
Proposal

This document is a concentration of 558 and incorporate feedback from the Amsterdam meeting.

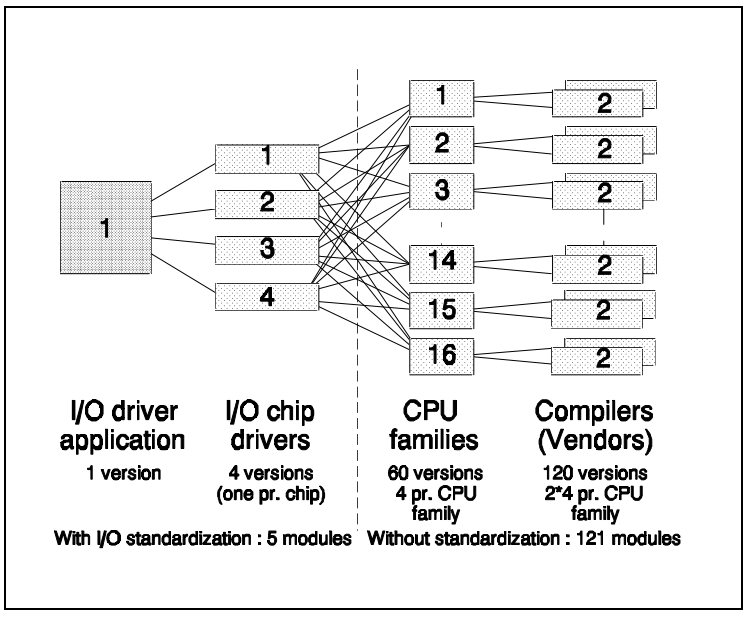
PURPOSE

Reasons for a standardization of functions for basic I/O hardware addressing:

1. Portability: If it is possible to reuse I/O hardware (I/O chips, I/O cells or I/O boards) with another platform, then it should also be possible to reuse the hardware device drivers for that I/O hardware in the new platform (without any modifications to the driver source code).

2. If there exist two C compilers from different vendors for the same platform, it should be possible to compile I/O hardware device driver source code with either compilers, and get the same (well defined) logical behaviors during runtime.

3. Reduce development and maintenance cost for multiplatform source code. The overhead associated with supporting drivers for multiple I/O chips, processor architectures, and compilers from different vendors, are reduced significantly when the C syntax for basic I/O operations on hardware is standardized.



In the above example an I/O application support 4 I/O chips, 16 of the most widely used CPU families on the market and approximately 2 compiler vendors pr. CPU family. By using the standardized syntax for basic I/O operations on hardware, as described in this proposal, the number of device driver modules to develop and maintain is reduced from 120 to 4, i.e. only one device driver for each I/O chip.

STANDARDIZATION METHOD:

The standardization of I/O addressing involves 3 components:

1. A few functions for making access to the most common I/O register sizes (register datawidths).
2. A standardized header file name: iohw.h
3. A new conceptional type: access_type

Synopsis example

```
#include <inttypes.h>
#include <iohw.h>

uint_8t iord8( access_type );        /* read byte */
void iowr8( access_type, uint_8t);   /* write byte */
void ioor8( access_type, uint_8t);   /* bit set */
void ioand8( access_type, uint_8t); /* bit clear */
```

A single parameter *access_type*, represent (or reference) a complete description of how a given I/O register is connected in a given hardware platform with a given processor architecture. This parameter enables the I/O function syntax to be standardized.

I/O registers have exact sizes, which are independent of how a compiler implements the standard data types. The standardized I/O functions therefore operate on unsigned fixed sized integral types as defined by the header file inttypes.h.

SEMANTICS FOR ACCESS_TYPE

The definition of *access_type* parameters will be implementation (compiler, processor and platform) specific. An I/O register definition should contain the following elements:

- A symbolic name for the I/O register

which represents:

- A complete specification of the access method for the I/O register in the given hardware platform:
  - I/O register data width: (i.e. 1, 8, 16, 32, 64 bits)
  - Address space used (if more than one address space)
  - Physical address (data size = register size)
    or
  - Physical address, bit position. (bit in register)

The *access_type* parameter definitions for the I/O registers in a given hardware platform is included via iohw.h.

THE HEADER FILE iohw.h

```
/* I/O functions for operation on a single register. */
/* For 8, 16, 32, 64 bit and 1 bit registers. */

/* Read operations: */
uint_8t iord8(access_type);
uint_16t iord16(access_type);
uint_32t iord32(access_type);
uint_64t iord64(access_type);
bool iordbit(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 iowrbit(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 ioandbit(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 ioorbit(access_type, bool);

/* I/O functions for operation 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 */

/* 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: */
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: */
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);
```

A conforming implementation may support an execution environment in which one or more of the functions descriped above cannot be implemented.

A conforming implementation may support an execution environment in which the *access_type* parameter in one or more of the functions above cannot be used as a dynamic variable.

EXAMPLE OF HOW TO USE THE I/O FUNCTIONS

```
/* Portable function for I/O chip with 16 registers */
#include <inttypes.h>
#include <iohw.h>   /* Define I/O functions and hardware */

void my_device(uint_16t data)
   {
   if (iord16(PORT1) & 0x4)     /* read */
       ioand16(PORT2, 0xfff0);  /* bit clear */
   else
       ioor16(PORT2, 0xC00);    /* bit set */
   iowr16(PORT3, data);         /* write */
   }
```

This driver function can be ported to any platform where the I/O chip can be connected. The driver function can be reused with the new without modifications to the driver source code itself. Only the iohw.h has to be replaced with a version to te new compiler / platform.

CONSTRAINTS

The I/O function should match the I/O register type

The correct I/O function to use with a given I/O register is defined by the capability of the I/O register. (Not by the addressing capability of the processor)

```
Use ioxx8 on 8 bit registers
Use ioxx16 on 16 bit registers
etc.
```

Constrains on allowed operations

The standardized I/O functions should NOT (try to) handle limitations which originate from limitations in the physical I/O hardware. (i.e. Hide the processor architecture differences, not the characteristics of the I/O hardware)

Example
AND and OR operations are not allowed
(are not possible) on the following register types:

```
write-only (WR)
read_only  (RD)
read_write (RD_WR)
```

The handling of limitations which originate from the I/O hardware registers must be done in the device driver source code. (Not by the standardized I/O functions).

Atomic operations

The standardized I/O functions do not guarantee read-modify-write operations will be atomic.

Example:

```
ioand8( PORT3, 0xf3 );
```

This statement may for some processor types be implemented as separate machine operations:

```
in temp,PORT3;
and temp,0xf3;
out PORT3,temp;
```

Compiler optimization

The standardized I/O functions do not guarantee two iordxx and iowrxx operations will always be two separate operations.
Two operations in a statement are allowed to be combined to a single read-modify-write operation by the compiler.

Example:

```
iowr8(PORT1, iord8(PORT1) + 0x10);
```

is allowed, if possible, to be optimized by the compiler to a single machine instruction:

```
ADD   PORT1,0x10
```

EXAMPLE: IMPLEMENTATION OF THE I/O FUNCTIONS

For processor architectures which only have one linear address space implementation of the I/O functions become very simple. An efficient implementation can be made using existing preprocessor directives:

```
// Defined in inttypes.h
#define uint_8t unsigned char

// Defined in iohw.h
#define access8 volatile uint_8t *

#define iowr8( PADDR, PVAL ) (* ( PADDR ) = ( PVAL ))
#define iowrbuf8( PADDR, PINDEX, PVAL ) \
                      (( PADDR )[( PINDEX )] = ( PVAL ))

#define PORT1 ((access8) (0xfff0)) // I/O register definition

// Portable driver functions
void nibblewr1 ( uint_8t data ) {
     iowr8( PORT1, data & 0xf);
     iowr8( PORT1, data >> 4);
     }

void nibblewr2 ( access8 adr, uint_8t data ) {
     iowr8( adr, data & 0xf);
     iowr8( adr, data >> 4);
     }

void nibblewr3 ( int index, uint_8t data ) {
     iowrbuf8( PORT1, index, data & 0xf);
     iowrbuf8( PORT1, index, data >> 4);
     }

void nibblewr4 ( access8 adr, int index, uint_8t data ) {
     iowrbuf8( adr, index, data & 0xf);
     iowrbuf8( adr, index, data >> 4);
     }

void iotest(void)
     {
     nibblewr1( 0x12 );
     nibblewr2( PORT1, 0x34 );
     nibblewr3( 10, 0x56);
     nibblewr4( PORT1, 3, 0x78 );
     }
```

COMMENTS TO 8 AND 16 BIT OPERATIONS
ON 32 AND 64 BIT MACHINES

If a machines only have 32 bit and 64 bit memory addressing operations, how can 8 and 16 bit I/O register operations then be implemented?

If I/O chips with 8 or 16 bit registers cannot be used in the hardware, then of course there will be no point in that the C compiler should implement 8 and 16 bit I/O functions.
(Note: This would normally not cause any portability problems either, simply because if 8 or 16 bit I/O chips cannot be used with the machines, there would be no point in reusing device driver source code for the I/O chips on those machines).

However, normally I/O chips with 8 or 16 bit registers would be connected to ex. the LSB part of the data bus and addressed with normal 64 / 32 bit machine operations as shown below.

Example:

An I/O chip with two 16 bit registers Y,Z connected to a 32 bit bus and addressed with 32 bit instructions.

The MSB part of the 32 value may then be masked by extra hardware which assure that the MSB part always returns 0.

```
                MSB                                 LSB
I/O register Z  00000000 00000000 ZZZZZZZZ ZZZZZZZZ
I/O register Y  00000000 00000000 YYYYYYYY YYYYYYYY
```

```
Or the MSB parts of a 32 value are don't care because they are
masked by the I/O function (the most common solution).
```

```
                MSB                                 LSB
I/O register Z  xxxxxxxx xxxxxxxx ZZZZZZZZ ZZZZZZZZ
I/O register Y  xxxxxxxx xxxxxxxx YYYYYYYY YYYYYYYY
```

Values for 8 and 16 I/O registers size are treated just like uint_8t and uint_16t data types would be.

Example: Implementation of 16 bit I/O functions for 32 / 64 bit machines

Implementing the standardized I/O functions for 16 bit I/O registers on a 32 / 64 bit machine with a single linear address space is done in nearly the same way as with smaller architectures.

```
// Definition from inttypes.h
#define access16  volatile uint_16t *

// Definition of hardware registers
#define PORT1 ((access16) ( 0xffffa00 ))
#define PORT2 ((access16) ( 0xffffa08 ))
#define PORT3 ((access16) ( 0xffffa10 ))

/* 16 bit I/O functions */
#define iord16( PADDR ) ((*( PADDR )) & 0xffff)

#define iowr16( PADDR, PVAL ) (*(PADDR) = ( PVAL ))

#define ioor16( PADDR, PVAL ) (*(PADDR) |= ( PVAL ))

#define ioand16( PADDR, PVAL ) (*(PADDR) &= ( PVAL ))


/* 16 bit I/O buffer functions */

#define iordbuf16( PADDR, PINDEX ) \
            (( PADDR[(PINDEX)] ) & 0xffff)

#define iowrbuf16( PADDR, PINDEX, PVAL ) \
            (PADDR[(PINDEX)] = ( PVAL ))

#define ioorbuf16( PADDR, PINDEX, PVAL ) \
            ( PADDR[(PINDEX)] |= ( PVAL ))

#define ioandbuf16( PADDR, PINDEX, PVAL ) \
            ( PADDR[(PINDEX)] &= ( PVAL ))
```

COMMENTS ABOUT DIFFERENT I/O ADDRESSING SCHEMES

Three different I/O addressing schemes for larger machines is sketched in the following:
1. With fixed address locations.
2. With relocatable chip-select base addresses.
3. With I/O page address mapping.

It is assumed that the machines in question have a single linear address space.

Ad.1 Fixed physical address locations
The I/O register are located on fixed absolute physical addresses in the given hardware platform. This is the most common way to locate I/O on smaller platform.
On larger platforms at least some of the system configuration registers will always be located on fixed addresses.

Fixed I/O register locations are defined as mentioned earlier. Example:
#define PORT1 (* (volatile uint_16t *) (0xffffa00))

Ad.2 Relocatable chipselect base addresses
The I/O chip is connected to a configurable chip select.

If a chipselect register in a given platform is always initiated with the same value, then the use of a chip select register just become a hardware implementation detail. All I/O registers connected to the chipselect line can be defined as with fixed physical addresses.
If the chipselect register is updated with different values during runtime, then a chip select base address must be included in the I/O definition:

```
volatile void *cs_base1;
#define PORT1 (* (volatile uint_16t *) (cs_base1+0xa00))
```

Ad.3 I/O page address mapping
When the device driver program is started the physical I/O addresses in question is mapped to a (usually smaller) virtual I/O page and in this way made accessible by the program.
The virtual to physical address conversion in the processors mapping hardware is totally transparent to the I/O device driver program.

The address locations defined in iohw.h should then be the I/O address locations within the virtual I/O page. I/O is defined just as with fixed address locations:

```
#define PORT1 (* (volatile uint_16t *) ( 0xa00 ))
```

In any of these cases the standardized I/O function syntax will hold.