Document Number: WG14 N771/X3J11

Title: GUIDELINES FOR IMPLEMENTING THE IOHW.H FUNCTIONALITY

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: Sept 23, 1997
Proposal Category:

- __ Editorial change/non-normative contribution
- __ Correction
- __ New feature
- __ Addition to obsolescent feature list
- __ Addition to Future Directions
- XX Other (please specify) Implementation guidelines

Area of Standard Affected:

- __ Environment
- __ Language
- __ Preprocessor
- __ Library
  - __ Macro/typedef/tag name
  - __ Function
  - __ Header
- __ Other (please specify) Information to implementors

______________________________

Prior Art: Widely used
Target Audience: Compiler vendors, Device driver programmers, I/O chip manufactures

Related Documents (if any):

| | |
|---|---|
| WG14 N509/X3J11 95-110 | Introduction paper |
| WG14 N558/X3J11 95-110 | Proposal |
| WG14 N656/X3J11 | Revised Proposal |
| WG14 N731/X3J11 | Revised Proposal |

-------------------------------

# Guidelines for implementing
# the IOHW.H functionality

(Draft 2.0)

## PURPOSE

The purpose with this document is to describe a complete set of  I/O register connections and addressing methods, which cover how most peripheral I/O chips are used with typical processor architectures. [F1]

This description is intended to be used as a guideline for implementing the header file IOHW.H and the functions defined herein.

A specific IOHW implementation for a given processor and platform architecture will typically use a defined subset of the I/O register connection and addressing methods described in this document. [F2]

Footnote 1:

In this document the term "I/O chips" cover : external I/O chips, internal I/O cells, plus I/O circuitry containing multiple chips with fixed relative chip addresses, all addressed in common way via a single width databus (i.e. typical plug-in boards).

Footnote 2:

Hardware constraints will typically limit the set of I/O addressing methods which can be used with a given processor architecture. For instance, connection of an I/O chip with 32 bit databus to a 16 bit processor architecture is considered to be non-applicable.

An implementor may also decide only to implement a specific subset of I/O addressing methods which he consider to be relevant for the his market place. For instance, an implementor may choose not to support addressing of 64 bit logical I/O registers if the IOHW implementation is for a 8 bit processor architecture.

# OVERVIEW

The list below shows topics and parameters which must be considered during an implementation of the IOHW.H functionality. For each topic selections must be made and decisions taken by the implementor. For processor platforms with multiple address ranges each range will typical require individual considerations.

Logical I/O registers: [F3]
- 1,8,16,32,64 bit
- Single physical register / multiple physical registers
- For bit addressing:
  - Bit position in register.

I/O chip databusses: [F4]
- Databus width: 1,8,16,32,64 bit
- MSB/LSB register order.

Processor databus widhts: [F5]
- Databus width: 8,16,32,64 bit
- Address alignment for adjacent I/O registers.

Addressing methods: [F6]
- Direct addressing:
  - Constant address
  - Constant address + constant offset
- Indirect addressing:
  - Variable based address
  - Variable based address + constant offset
- Expanded addressing:
  - Direct I/O function call.
  - Indirect I/O function call.

Machine level considerations

Footnote 3:

The logical register width equals (or is less than) the size of the data variable used in the I/O function headers defined by IOHW.H.
The logical register can be looked upon as the operational register unit. For addressing purposes the logical register may have been split in multiple physical registers of lesser size.

Footnote 4:

The I/O chip databus width is the number of physical data bit lines out of the I/O chip. The width of the databus may be less than the logical register width.

Footnote 5:

The processor databus width is the logical bus width at the point where the I/O circuitry is connected. The internal (on-chip) processor databus width is often larger than the external (off-chip) databus width. The logical processor databus width may vary, and may be different for different address ranges.

Footnote 6:

Direct addressing with a constant address seems to be supported by all processor architectures on all address ranges. With many processor architectures indirect addressing must be implemented purely in software. A few processor architectures do not support indirect addressing at all on some of their address ranges (i.e. the address need to be a part of the machine instruction).

**Limitations**

Not all parameter combinations in the list above are possible. Natural hardware limitations will simplify the IOHW implementation. The table below illustrates which combinations of processor databus widths and I/O chip databus widths are possible:

| Processor | Databus width for I/O chip | | | |
|---|---|---|---|---|
| databus width | 8 bit | 16 bit | 32 bit | 64 bit |
| 8 bit | Possible | NA | NA | NA |
| 16 bit | Possible | Possible | NA | NA |
| 32 bit | Possible | Possible | Possible | NA |
| 64 bit | Possible | Possible | Possible | Possible |

**Bit addressing**.

Single bit addressing, with a one-bit databus and a linear bit address range, are only supported by a few processor architectures and typically only with internal (on-chip) I/O registers.
In all other cases bit addressing is equal to addressing a bit in a I/O register of a larger size. Seen from the I/O register the basic addressing operations used for bit addressing are therefore the same at used for addressing of 64-8 bit registers.

The bit addressing case can therefore be omitted as an individual case in most of the following considerations.

## LOGICAL I/O REGISTER CONSIDERATIONS

The width of a logical I/O register is considered always to be equal to, or larger than, the width of the databus for the I/O chip.

In cases where the logical register width is larger than the databus width, the I/O chip must be accessed with a number of addressing cycles, as shown in the table below.

| Logical I/O | Databus width for I/O chip | | | |
|---|---|---|---|---|
| register width | 8 bit | 16 bit | 32 bit | 64 bit |
| 8 bit | 1 a.c. | NA | NA | NA |
| 16 bit | 2 a.c. | 1 a.c. | NA | NA |
| 32 bit | 4 a.c. | 2 a.c. | 1 a.c. | NA |
| 64 bit | 8 a.c. | 4 a.c. | 2 a.c. | 1 a.c. |

## Implementation

All implementations should support the single-cycle register access cases (for register widths supported by the processor architecture). That is, all cases where the logical I/O register width equals the I/O chip databus width.

An implementor may decide that the multi-access-cycle cases are special cases which should be handled at the device driver level, for instance as illustrated in the following examples:

**Example 1:**

```
/* Read a logical 32 bit register in an I/O chip with 8 bit databus */
uint_least32_t  readcounter(void)
   {
   uint_least32_t  val;
   val = (uint_least32_t) iord8(COUNT0);
   val += (uint_least32_t) iord8(COUNT1) << 8;
   val += (uint_least32_t) iord8(COUNT2) << 16;
   val += (uint_least32_t) iord8(COUNT3) << 24;
   return val;
   }
```

**Example 2:**

```
/* A much faster implementation uses a union
   for type conversion */
uint32_t  readcounter(void)
   {
   union {
      uint32_t  ul;
      struct {
         uint8_t b0;
         uint8_t b1;
         uint8_t b3;
         uint8_t b4;
         } b;
   } val;
   val.b.b0 = iord8(COUNT32_0);
   val.b.b1 = iord8(COUNT32_1);
   val.b.b2 = iord8(COUNT32_2);
   val.b.b3 = iord8(COUNT32_3);
   return val.ul;
   }
```

Example 1 is fully portable, but has a large machine code overhead.
Example 2 is faster, but usually non-portable, because it depends heavily on the size of the data types and the data alignment for the given compiler/linker.

The above examples illustrate why it is highly recommended that the multi-access-cycle cases for logical I/O registers are supported by the IOHW implementation.

```
/* iord32() can hide the multibel address cycles with associated
   large logical I/O registers in small chips */
uint_least32_t  val;
val = iord32(COUNT32);
```

The I/O multi-access-cycles are usual good candidates for optimization at compiler level.

# I/O CHIP DATABUS CONSIDERATIONS

## MSB-LSB addressing order for I/O registers

When an I/O chip contains a register with a logical width larger than the width of the I/O chip databus, the MSB/LSB addressing order become relevant. [F7]

Footnote 7:

Do not confuse the MSB-LSB addressing order for registers in an I/O chip, with the high-endian/low-endian for the processor architecture. Those are totally independent parameters.

**Example**:
16 bit logical register in a chip with a 8 bit chip databus:

Ascending register order:

```
<LSB, bit  7-0>    Address + 0
<MSB, bit 15-8>    Address + n
```

Descending register order:

```
<MSB, bit 15-8>    Address + 0
<LSB, bit  7-0>    Address + n
```

Where **n** is the address alignment constant for the I/O register connection. **n** will typically be either 0,1,2,4 or 8, with different databus width combinations. [F8]

Footnote 8:

The value 0 for **n** is the special case where an I/O chip requires multiple access cycles at the same address to get access to all sub-elements of  the logical register.

# PROCESSOR DATABUS

When the logical width of the processor databus, at the point where the I/O chip is connected, is larger than the I/O chip databus width then the address alignment parameter become relevant.

The address alignment parameter define the address distance between adjacent I/O registers. The parameter is used both with the *ioxxbufn(..)* functions and with logical I/O registers requiring multiple access cycles.

**Example**
An I/O chip with 8 bit data bus connected to the LSB part of a 16 bit processor databus with a byte aligned address range. The address alignment value equals 2.

```
Processor bus:           <15  -  8, 7   -  0>
Register element 0        XXXXXXXX <00000000>     Address + 0*2
Register element 1        XXXXXXXX <11111111>     Address + 1*2
Register element 2        XXXXXXXX <22222222>     Address + 2*2
Register element 3        XXXXXXXX <33333333>     Address + 3*2
```

The table below shows the typical value for the address alignment parameter with different chip databus / processor databus combinations, where the processor has a byte aligned address range.

| Logical processor | Databus width for I/O chip | | | |
|---|---|---|---|---|
| bus width | 8 bit | 16 bit | 32 bit | 64 bit |
| 8 bit | 0, +1 | NA | NA | NA |
| 16 bit | +2 | 0, +1 | NA | NA |
| 32 bit | +4 | +2 | 0, +1 | NA |
| 64 bit | +8 | +4 | +2 | 0,+1 |

If a processor architecture only support for instance a 32 bit aligned address range, then the 8 bit and 16 operations must be implemented as masked 32 bit operations. In this case the address alignment table would look like this:

| Logical processor | Databus width for I/O chip | | | |
|---|---|---|---|---|
| bus width | 8 bit | 16 bit | 32 bit | 64 bit |
| (8 bit) | 0,+1 | NA | NA | NA |
| (16 bit) | 0,+1 | 0,+1 | NA | NA |
| 32 bit | 0,+1 | 0,+1 | 0,+1 | NA |
| 64 bit | +2 | +2 | +2 | 0,+1 |

## Logical databus versus physical databus

Note that the "logical" databus width for the processor can be smaller than the physical databus width. With some processor architectures the bus management unit is able to use only a subset of physical databus lines when addressing I/O registers of a lesser size.

**Example**
An I/O chip with 8 bit data bus connected to the LSB part of a 16 bit processor databus. The LSB bus part is configurated as a logical 8 bit databus with byte aligned address range. The value for the address alignment parameter then equals 1.

```
Processor bus:           (<15 - 7>),<7  -   0>
Register element 0        --------  <00000000>    Address + 0*1
Register element 1        --------  <11111111>    Address + 1*1
Register element 2        --------  <22222222>    Address + 2*1
Register element 3        --------  <33333333>    Address + 3*1
```

An implementation should allow the user to define the address alignment parameter individually for each I/O register, whenever the address alignment parameter is needed.

## ADDRESSING METHODS

The following addressing methods are recommended and are able to cover the I/O addressing requirements with most platforms:

```
DIRECT ADDRESSING:              CONSTANT ADDRESS
DIRECT ADDRESSING:              CONSTANT ADDRESS + CONSTANT OFFSET

INDIRECT ADDRESSING:            VARIABLE BASED ADDRESS
INDIRECT ADDRESSING:            VARIABLE BASED ADDRESS + CONSTANT OFFSET

EXPANDED ADDRESSING:            DIRECT I/O FUNCTION CALL.
EXPANDED ADDRESSING:            INDIRECT I/O FUNCTION CALL.
```

### Direct addressing

With direct addressing the address value is defined as a constant.
Direct addressing seems to be supported by all processors and all address ranges, and is usually the fastest way to make I/O register access. Often the I/O register address will be a part of the machine instruction.

Direct addressing is convenient when the hardware architecture design is stable and the I/O register address is well-defined and constant.

**Example**

```
// Address for chip X (platform dependent)
#define CHIPXADR     0x1000

// Offset definement (chip dependent, platform independent)
#define REG_A_ADR  (CHIPXADR+0)  // Register offset base
#define REG_B_ADR  (CHIPXADR+1)  // Register offset 1
#define REG_C_ADR  (CHIPXADR+3)  // Register offset 3
```

### Indirect addressing

With indirect addressing the base address value is located in a variable. The I/O address will be fetched at runtime.

Indirect addressing is convenient when the hardware configuration may vary. The base addresses is assumed to be assigned by another part of the system before use of the IOHW functions.
Portability note: A few processor architectures are not able to support indirect addressing on some address ranges.

**Example**

```
unsigned int IOslot_A    // Slot address assigned by system
                          //  before any use of I/O
// Address for chip X in slot A (platform dependent)
#define CHIPXADR     IOslot_A+32
// Offset definement (chip dependent, platform independent)
#define REG_A  (CHIPXADR+0)   // Register offset base
#define REG_B  (CHIPXADR+1)   // Register offset 1
#define REG_C  (CHIPXADR+3)   // Register offset 3
```

## Expanded addressing

With expanded addressing the address value and data values are redirected to user-defined I/O addressing functions.
Expanded addressing is a convenient way to expand the platform hardware architecture with virtual address/data busses or bridged busses, in a way which are transparent to the I/O driver source code.

With direct expanded addressing the user defined functions must be known at compile time. The user defined function can be a macro.
With indirect expanded addressing the user defined functions are called via pointers to functions, and the user defined function can be assigned at runtime.

In both cases the data parameter passed is equals the data parameter for the IOHW function, and the address parameter is typical the resulting efficient address as given by the IOHW implementation for the given address bus range.

**Example**

```
/* User defined functions for platform expansion with virtual I/O busses.
   The functions are invoked via IOHW functions */

void virtual_bus_wr16( uint16_t address, uint16_t dat)
   {
   iowr16(BRIDGE_ADDR,address);  // Set virtual address lines
   iowr16(BRIDGE_DATA, dat);     // Write data to virtual bus
   }

uint16_t virtual_bus_rd16( uint16_t address)
   {
   iowr16(BRIDGE_ADDR,address);  // Set virtual address lines
   return iord16(BRIDGE_DATA);   // Read data from virtual bus
   }
```

## SOME MACHINE-LEVEL CONSIDERATIONS

**AND and OR operations on I/O registers**
Usually the I/O chip hardware itself only support the basic hardware operations READ and WRITE.

Operations like AND and OR are performed as combined 3 step operations READ-operation-WRITE, where the AND or OR operation itself must take place in the processor hardware.
It is recommended that AND and OR operations on I/O registers are implemented as one indivisible (atomic) operation whenever possible.

**Atomic operation using read-modify-write instructions**
If the processor architecture supports AND and OR operations as a single read-modify-write operation for the address bus in question, then such read-modify-write operations should be used.

**Atomic operation using special instructions**
If the processor architecture does not support AND and OR operations as read-modify-write operations for the address range, but have some special instructions to assure atomic operations for a number of succeeding instruction, then atomic operations should be implemented.

**Non-volatile I/O addresses**
With indirect addressing an implementation can typically assume that the variable base address behaves like a non-volatile value. The I/O base address therefore does not necessarily need to be reloaded from memory during every I/O operation. With multi-access-cycles this may increase the efficiency significantly.