# TECHNICAL REPORT
# ON
# BASIC I/O HARDWARE ADDRESSING

C++ performance group
Draft

## 1 Scope

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 in the language, and today almost all C compilers for freestanding environments and embedded systems support direct access to I/O hardware registers from the C source level. However, these extensions have not been consistent across dialects. As a growing number of C++ compiler vendors are now entering the same market, the same I/O driver portability problems become apparent for C++.

This Technical Report is a step towards codifying common existing practice in the market and providing a single uniform syntax for basic I/O hardware register addressing.

## 2 Rationale

*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 get the same logical behavior at 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 problem areas are the same for C and C++, and the standardization method proposed is applicable for both languages.

## 2.1 Standardization objectives

### 2.1.1 Basic requirements

A standardization method for basic I/O hardware addressing must be able to fulfill three requirements at the same time:

**S** The standardized syntax must not prevent compilers from producing machine code with 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.

**S** The I/O driver source code modules should be completely portable to any processor system (from 8-bit systems and up) without needing any modification to the driver source code itself. I.e. the syntax should promote *I/O driver source code portability* across different execution environments.

**S** 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.)

### 2.1.2 New perception of I/O registers simplifies the syntax standardization.

There have been several different attempts to create an international standardfor the general syntax for basic I/O operations over the years, but these all failed to meet the special requirements of the embedded market and the market for freestanding environments.

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

*The I/O standardization method used here overcomes 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.*

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

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

### 2.1.3 Typical I/O register characteristics

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 processor architectures and bus systems.

Most important is the fact that I/O registers do not usually 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 the default should be that I/O registers be treated as *volatile* data types.

### 2.1.4 Access method encapsulation

Since processor architectures and hardware platforms ARE different, standardization must also provide a method of separating the description of the hardware differences and addressing methods from the source code. The standardization method should *encapsulate* descriptions of

hardware differences, e.g. 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 using a few standardized I/O *functions (*or *class member functions)*. This corresponds to the way encapsulation is done in the spirit of C/C++. (The functions may be implemented as in-line functions, macros, or templates for speed optimization.)

### 2.1.5 Reduced basic operation set

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 a standard only has to define functions for the most basic operations on I/O registers.

A standardization method must define basic *read* and *write* operations as a minimum. In addition the iohw header includes functions for the most common I/O register operations: set, clear and invert for one or more register bits.

The programmer can build all other arithmetic and logical operations on top of these few basic I/O access operations.

### 2.1.6 Handling of intrinsic types

With many existing processor architectures, I/O register access often requires the use of special machine instructions to operate on special I/O address ranges.

This means that an extension of the type system is needed so that I/O registers can be accessed from the C/C++ source level. If a *function syntax* is used for standardized I/O access, all use of processor and platform specific I/O access types (implementation specific types) will be limited 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 that is portable to any processor system, without extending the type system defined by the C/C++ standard.

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

# 3 Basic concepts

## 3.1 Simple conceptual model for I/O registers

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

*Example:*

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

The programmer only sees the characteristics of the I/O register itself. The underlying platform, bus architecture, and compiler implementation do not matter during driver programming. The underlying system hardware may later be changed without modifications to the I/O driver source code being necessary.

## 3.2 The access_type parameter

A new *access_type* type is used by the standardized I/O functions.

Example

```
uint_8t iord8(access_type);            // Read from I/O register
void iowr8(access_type, uint_8t);      // Write to I/O register
```

*The* access_type *parameter represents or references a complete description of how the 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. The definition of an I/O register object may or may not require a memory instantiation, depending on how a compiler vendor has chosen to implement *access_types*. For maximum performance this could be a simple definition based on compiler specific address range and type qualifiers, in which case no instantiation of an *access_type* object would be needed in data memory.

Footnote: 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 avoids telling how the underlying file system should be implemented or initialized.

## 3.3 Exact sized data types

The data parameter and return parameters used in the I/O functions are integer data types with an exact (minimum) bit precision.

I/O registers have a fixed size independent of how a compiler implements the standard integer

types. Data values for use with I/O registers should therefore always have an exact (minimum) size independent of the compiler implementation.

Another reason for the exact-sized types is to allow the programmer to decide what precision is needed by the 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 a program uses integer data types with a precision greater than needed by the application. Fixed sized data types are therefore a performance issue and not just related to I/O.

The exact-sized data types are currently defined in the header file <**stdint.h**> (ISO/IEC 99 Programming Language C). It is suggested that these types be adopted by C++.

## 3.4 I/O initialization

With respect to the standardization process it is important to make a clear distinction between I/O hardware (chip) related initialization and platform related initialization. Typically three types of initialization are 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) is relevant for basic I/O hardware addressing.

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.

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

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

Note, this means 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 creates performance problems with implementations for C99 compilers). Apart from this performance issue, I/O selector initialization is not a problem with respect to basic I/O hardware addressing.

I/O access initialization concerns the initialization and definition of *access_type* objects.
This process is implementation defined. It depends both on the platform and processor architecture and on which underlying access methods are supported by an *iohw* implementation.

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 is often allocated dynamically at runtime. Here the *access_type* information can only be partly defined at compile time. Some platform software dependent part of it must be initialized at runtime.

When designing the *access_type* object a compiler implementer 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 need to contain dynamic information, the *access_type* object may or may not require an instantiation in data memory. If more of the information is static, a better execution performance can usually be achieved.

See the Guide for implementers for further discussion of different access methods.

# 4 The <iohw> header

## 4.1 Overview

The header file **<iohw>** defines a number of functions which:

- Support the most common fixed register sizes.
  - 8-bit, 16-bit, 32-bit, 64-bit or 1-bit (logical)
- Support the most basic I/O register operations.
  - Read, Write,
  - Bit-set (Or) in register, bit-clear (And) in register, bit-invert (Xor) in register.
  - Single register objects, register array objects.
- Define new abstract types for I/O register referencing: *access_type*(s)
- Provide a uniform encapsulation method for hardware and platform differences.
- Provide a uniform header file name. <iohw>

## 4.2 Single register access

The I/O access functions defined here are for operations on a single register object. The functions are defined 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);

/* XOR operations (Invert group of bits) */
void ioxor8(access_type_8, uint_8t);
void ioxor16(access_type_16, uint_16t);
void ioxor32(access_type_32, uint_32t);
void ioxor64(access_type_64, uint_64t);
void ioxor1(access_type_1, bool);
```

The one-bit functions (ioxxx1) access a single bit in a register. The access_type_1 parameter defines both how to access the I/O register and the position of the bit in the register

## 4.3 Register array access

This refers to I/O functions for operations on I/O register array objects. It could be I/O circuitry with internal buffers or multiple registers, e.g. a peripheral chip with a linear hardware buffer.

The *index* parameter is 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);

/* XOR operations on hardware buffers (Invert group of bits) */
void ioxorbuf8(access_type_8, unsigned int index, uint_8t dat);
void ioxorbuf16(access_type_16, unsigned int index, uint_16t dat);
```

```
void ioxorbuf32(access_type_32, unsigned int index, uint_32t dat);
void ioxorbuf64(access_type_64, unsigned int index, uint_64t dat);
```

In addition to these I/O function definitions, *iohw.h* must also contain definitions for *access_types* and must define the fixed sized types (perhaps by including <**stdint.h**>)

## 4.4 Common function syntax for C and C++

It would be beneficial, especially in the market for freestanding environments and embedded systems, if source code for I/O hardware drivers could be written so that the driver code can be compiled with both C and C++ compilers. In this market C compilers are still dominant.

With a C-like syntax for basic I/O hardware access, users could benefit from a broader range of source library products from third party vendors. A common syntax would also ensure a smoother transition in this market from C to C++.

Therefore it is recommended that the same C function syntax for basic I/O hardware addressing be used with both C and C++.

Note that it is only the standardized function interfaces that need to be C-like. *iohw* and *access_type* implementations for C and C++ may be very different in both performance and the number of access methods supported. An implementation for C++ can still take advantage of templates, classes and other advanced C++ features.

# Appendix X
## Implementing the iohw header
A guide for implementers

## X.1 Purpose

The *iohw* header defines a standardized function syntax for basic I/O hardware addressing. This header should normally be created by the compiler vendor.

The idea behind the standardized syntax is that the source code looks the same independent of where the I/O register is located in the hardware and independent of the underlying method used to address the I/O hardware register.

While this standardized function syntax for basic I/O hardware addressing provides a simple, easy-to-use method for a programmer to write portable and hardware-platform-independent I/O driver code, nevertheless the iohw header implementation itself may require careful consideration to achieve an efficient implementation.

This chapter gives some guidelines for implementers on how to implement the *iohw* header in a relatively straightforward manner given a specific processor and bus architecture.

### X.1.1 Recommended steps

Briefly, the recommended steps for implementing the *iohw* header are:

**S** Get an overview of all the possible and relevant ways the I/O register hardware is typically connected with the given bus hardware architectures, and get an overview of the basic software methods typically used to address such I/O hardware registers.

**S** Define a number of I/O functions, macros and *access_types* which support the relevant I/O access methods for the given compiler market.

**S** Provide a way to pick the right I/O function at compile time and generate the right machine code based on the *access_type* type or the *access_type* value.

### X.1.2 Compiler considerations

In practice an implementation will often require that very different machine code is generated for different I/O access cases. Furthermore, with some processor architectures, I/O hardware access will require the generation of special machine instructions not used otherwise when generating code for the traditional C, C++ memory model.

Selection between different code generation possibilities must be determined solely by the *access_type* declaration for each I/O register. Whenever possible this access method selection should be implemented so it is done entirely at compile time, in order to avoid any runtime or machine code overhead.

For a compiler vendor, of course, selection between code generation possibilities can always be implemented by supporting different intrinsic *access_type* types and keywords designed specially for the given processor architecture and additional to the traditional types and keywords defined by the language.

However, with a conforming C++ compiler, an efficient and all-round implementation of the *iohw* header can usually be made using template functionality. A template solution allows the number of compiler specific intrinsic I/O types or intrinsic I/O functions to be minimized or even removed completely, depending on the processor architecture.

For compilers not supporting templates (such as C compilers) other implementation methods must be used. In any case, at least the most basic *iohw* functionality can be implemented efficiently using a mixture of macros, *in-line* functions and intrinsic types or functions. Full feature *iohw* implementations will usually require direct compiler support (or extensions to the language).

The considerations described in the following are generally applicable to both C and C++ compilers. However, in this document it is primarily C++ template-based solutions that are used in the implementation examples.

## X.2 Overview of I/O hardware connection options

The various ways an I/O register can be connected to processor hardware are primarily determined by combinations of the following three hardware characteristics:

1. The bit width of the logical I/O register.
2. The bit width of the data-bus of the I/O chip.
3. The bit width of the processor-bus.

### X.2.1 Multi-addressing and I/O register endian

If the width of the logical I/O register is greater than the width of the I/O chip data bus, an I/O access operation will require multiple consecutive addressing operations.

The I/O register endian information describes whether the MSB or the LSB byte of the *logical I/O register* is located at the *lowest* processor bus address.
(Note that the I/O register endian has nothing to do with the endian of the underlying processor hardware architecture).

**Table: Logical I/O register / I/O chip addressing overview**

| **Logical I/O register widths** | **I/O chip bus widths** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | *8-bit chip bus* | | *16-bit chip bus* | | *32-bit chip bus* | | *64-bit chip bus* | |
| | LSB-MSB | MSB-LSB | LSB-MSB | MSB-LSB | LSB-MSB | MSB-LSB | LSB-MSB | MSB-LSB |
| *8-bit register* | direct | | n.a. | | n.a. | | n.a. | |
| *16-bit register* | r8{0-1} | r8{1-0} | Direct | | n.a. | | n.a. | |
| *32-bit register* | r8{0-3} | r8{3-0} | r16{0-1} | r16{1-0} | Direct | | n.a. | |
| *64-bit register* | r8{0-7} | r8{7-0} | r16{0,3} | r16{3,0} | R32{0,1} | r32{1,0} | Direct | |

(For byte-aligned address ranges)

### X.2.2 Address Interleave

If the size of the I/O chip data bus is less than the size of the processor data bus, buffer register addressing will require the use of *address interleave*.

**Example:**

If the processor architecture has a byte-aligned addressing range and a 32-bit processor data bus, and an 8-bit I/O chip is connected to the 32-bit data bus, then three adjacent registers in the I/O chip will have the processor addresses:

<addr + 0>, <addr + 4>, <addr + 8>

This can also be written as

<addr + *interleave**0>, <addr+*interleave**1>, <addr+*interleave**2>

where *interleave* = 4.

**Table: Interleave overview:** (bus to bus interleave relations)

| **I/O chip bus widths** | **Processor bus widths** | | | |
|---|---|---|---|---|
| | *8-bit bus* | *16-bit bus* | *32-bit bus* | *64-bit bus* |
| *8-bit chip bus* | Interleave 1 | interleave 2 | Interleave 4 | interleave 8 |
| *16-bit chip bus* | n.a. | interleave 2 | Interleave 4 | interleave 8 |
| *32-bit chip bus* | n.a. | n.a. | Interleave 4 | interleave 8 |
| *64-bit chip bus* | n.a. | n.a. | n.a. | interleave 8 |

(For byte-aligned address ranges)

### X.2.3 I/O connection overview:

The two tables above can be combined and will then show all relevant cases for how I/O hardware registers can be connected to a given processor hardware bus.

**Table: Interleave between adjacent I/O registers in buffer** (all cases).

| ***I/O Register width*** | ***Chip bus*** | | | ***Processor data bus width*** | | | |
|---|---|---|---|---|---|---|---|
| | *Bus width* | *LSB MSB* | *No. Opr.* | *width=8* | *width=16* | *width=32* | *width=64* |
| | | | | *size 1* | *size 2* | *size 4* | *size 8* |
| *8-bit* | *8-bit* | *n.a.* | *1* | **1** | **2** | **4** | **8** |
| *16-bit* | *8-bit* | *LSB* | *2* | **2** | **4** | **8** | **16** |
| | | *MSB* | *2* | **2** | **4** | **8** | **16** |
| | *16-bit* | *n.a.* | *1* | n.a. | **2** | **4** | **8** |
| *32-bit* | *8-bit* | *LSB* | *4* | **4** | **8** | **16** | **32** |
| | | *MSB* | *4* | **4** | **8** | **16** | **32** |
| | *16-bit* | *LSB* | *2* | n.a. | **4** | **8** | **16** |
| | | *MSB* | *2* | n.a. | **4** | **8** | **16** |
| | *32-bit* | *n.a.* | *1* | n.a. | n.a. | **4** | **8** |
| *64-bit* | *8-bit* | *MSB* | *8* | **8** | **16** | **32** | **64** |
| | | *LSB* | *8* | **8** | **16** | **32** | **64** |
| | *16-bit* | *LSB* | *4* | n.a. | **8** | **16** | **32** |
| | | *MSB* | *4* | n.a. | **8** | **16** | **32** |
| | *32-bit* | *LSB* | *2* | n.a. | n.a. | **8** | **16** |
| | | *MSB* | *2* | n.a. | n.a. | **8** | **16** |
| | *64-bit* | *n.a.* | *1* | n.a. | n.a. | n.a. | **8** |

(For byte-aligned address ranges)

### X.2.4 Generic buffer index

The interleave distance between two logically adjacent registers in an I/O register array can be calculated from:

1. The size of the logical I/O register in bytes.
2. The processor data bus width in bytes.
3. The chip data bus width in bytes.

Conversion from I/O register index to address offset can be calculated using the following generic formula:

```
Address_offset = index *
                  sizeof( logical_IO_register ) *
                   sizeof( processor_data_bus ) /
                    sizeof( chip_data_bus )
```

where a byte-aligned address range is assumed, the widths are a whole number of bytes, the width of the l*ogical_IO_register* is greater than or equal to the width of the *chip_data_bus*, and the width of the *chip_data_bus* is less than or equal to the *processor_data_bus*.

## X.3 Exact sized data types

I/O registers have exact bit widths which are independent of the compiler integer implementations and the integer types supported by the processor hardware.

An *iohw* implementation should therefore define exact width types for at least 8, 16, 32 and 64-bit registers. The definition should map the exact width integer types to an unsigned integer type of a matching width given by the compiler implementation (and optional compilation modes).

The exact width types can be defined directly in the *iohw* header. Alternatively the *iohw* can include the C99 standard header *stdint.h.*

**Example:**

```
// Definition of exact sized data types for the compiler.
// (In C9X these types are defined by the header stdint.h)
#define uint8_t  unsigned char
#define uint16_t unsigned short
#define uint32_t unsigned long
#define uint64_t unsigned long long
```

### X.3.1 Other register sizes

The most common widths of I/O chip busses and I/O registers are 2^N. However, if other bus and I/O register widths are native for a given processor architecture (for instance 24 bits) a vendor is free to extend an *iohw* implementation and incorporate functions for any odd-sized register widths along the line: iord24(..), iowr24(..), etc.

Footnote: As such processor-specific I/O registers are hardly ever used in other platforms, cross-platform portability will seldom be an issue for driver code which uses such iord24(..), iowr24(..) functions.

## X.4 Implementation of the access type parameter:

The number of I/O access functions which must be implemented in order to cover all connection cases depends on how the access type parameter is constructed. As shown in the following table an implementation can, with advantage, have at least the processor bus width as a separate parameter:

**Table: Number of access functions for each access method**.

| *Data bus widths supported by processor* | *8-bit* | *8–16-bit* | *8–32-bit* | *8–64-bit* |
|---|---|---|---|---|
| *Total number of access cases* | 7 | 19 | 34 | 50 |
| *Number of access functions to implement when using a processor bus size parameter* | 7 | 12 | 15 | 16 |

The possible I/O chip connections can be described with a single parameter which combines information about the I/O chip data bus width and the I/O register endian. The possible I/O register to bus connections can therefore be completely specified using only two parameters:

**S** A bus parameter which specifies access relations between the I/O chip data bus and the processor data bus.

**S** A multi-addressing and endian parameter which specifies access relations between the logical I/O register and the I/O chip data bus.

**Example 1:**

Definition of general I/O register connection types:

```
typedef enum {bw8 = 1, bw16 = 2, bw32 = 4, bw64 = 8} bus_t;
typedef enum {chip8, chip8l, chip8h, chip16, chip16l, chip16h,
              chip32, chip32l, chip32h, chip64} chip_t;
```

**Example 2:**

Possible I/O register connections with the processor H8/300H
(supporting only an 8-bit and a 16-bit processor data bus)

```
typedef enum {bw8 = 1, bw16 = 2} bus_t;
typedef enum {chip8, chip8l, chip8h, chip16, chip16l, chip16h} chip_t;
```

**Example 3:**

Template implementation of *access_type* for direct addressing of memory-mapped I/O registers:

```
typedef uint32_t address_t; // Address range type.

// Define an access_type template for direct addressing, combining
// IO register width, I/O address, processor bus width, chip bus width
// and endian

template <class T, address_t address, bus_t buswidth, chip_t chiptype>
class IO_MM { };


//--- Use of access_type in I/O register declarations made by user
//--- (normally placed in a separate platform dependent header file)

// an 8-bit register in an 8-bit chip using  8-bit processor bus mode.
typedef IO_MM <uint8_t,  41000, bw8, chip8>  PORT1;
// a 32-bit register in an 8-bit chip using 16-bit processor bus mode
typedef IO_MM <uint32_t, 41800, bw16, chip8l> PORT2;
```

### X.4.1 Access_types for different processor busses

If the processor architecture has multiple different addressing ranges (i.e. it requires different sets of machine instructions for the different busses), each addressing range should have its own set of *access_type* specifications.

**Example:**

The 80x86 processor architecture has two addressing ranges: the normal memory bus with memory-mapped I/O and a separate bus for I/O only. Implementations for the 80x86 processor

family will therefore require at least two sets of *access_type* specifications.

```
typedef uint32_t address_t; // Memory-mapped address range type.
typedef uint16_t io_addr_t; // IO-mapped address range type.


template <class T, address_t address, bus_t buswidth, chip_t chiptype>
   class IO_MM { };
template <class T, io_addr_t address, bus_t buswidth, chip_t chiptype>
   class IO_IOM { };
```

### X.4.2 Access_types for different I/O addressing methods

The following typical addressing methods should be considered by an implementer:

**S** *Address is defined at compile time.*
The address is a constant. This is the simplest case and also the most common case with smaller architectures.

**S** *Base address initiated at runtime.*
Variable base address + constant offset. I.e. the *access_type* must contain an address pair (address of base register + offset address).

The user-defined base address is normally initialized at runtime (by some platform-dependent part of the program). This also enables a set of I/O driver functions to be used with multiple instances of the same I/O hardware.

**S** *Indexed bus addressing*
Also called orthogonal or pseudo-bus addressing. It is a common way to connect a large number of I/O registers to a bus, while still only occupying a few addresses in the processor address space.
This is how it works: First the index address (or pseudo-address) of the I/O register is written to an address bus register located at a given processor address. Then the data read/write operation on the pseudo-bus is done via the following processor address. I.e. the *access_type* must contain an address pair (the processor address of indexed bus, and the pseudo-bus address (or index) of the I/O register itself).

This access method also makes it particularly easy for a user to connect common I/O chips, which have a multiplexed address/data bus, to a processor platform with non-multiplexed busses using a minimum amount of glue logic. The driver source code for such an I/O chip is then automatically made portable to both types of bus architecture.

**S** *Access via user-defined access driver functions.*
These are typically used with larger platforms and with small single chip processors (e.g. to emulate an external bus). In this case the *access_type* must contain pointers or references to access functions.

The access driver solution makes it possible to connect a given I/O driver source library to any kind of platform hardware and platform software using the appropriate platform-specific interface functions.

In general an implementation should always support the simplest addressing case; whether it is the constant address or base address method that is used will depend on the processor

architecture. Apart from this, an implementer is free to add any additional cases required to satisfy a given market.

Because of the different number of parameters required in an *access_type* specification and because of the different parameter ranges used, it is often convenient to define a number of different a*ccess_type* formats for the different access methods.

For the same reasons it is often convenient to implement the *iord1, iowr1, ioor1, ioand1,* and *ioxor1* functions so that they use their own *access_type* format, simply because of the extra parameters needed to specify the bit position in the register.

**Example:**

```
// Define types used in access_type declarations
typedef uint32_t address_t;       // Memory mapped address range
typedef uint8_t  sub_address_t;   // Sub address on indexed bus
typedef uint16_t io_addr_t;       // User I/O driver address
typedef uint8_t  bit_pos_t;       // Bit position in register

// Define access_type template for direct addressing
template <class T, address_t address, bus_t buswidth, chip_t chiptype >
     class IO_MM { };

// Define access_type template for addressing via base register
template <class T, address_t * base, address_t offset,
    bus_t buswidth, chip_t chiptype >
     class IO_MM_BASE { };

// Define access_type template for indexed bus addressing
template <class T, address_t address, sub_address_t idx,
    bus_t buswidth, chip_t chiptype >
     class IO_MM_IDX { };

// Define access_type for user-supplied access driver functions
template <class T, io_addr_t address,
  T iord( io_addr_t address), void iowr( io_addr_t address, T val) >
     class IO_MM_DRV { };

//  Define access_type for direct addressing of bit in register
template <class T, address_t address, bit_pos_t bitpos,
  bus_t buswidth, chip_t chiptype >
     class IO_MM_BIT { };
```

### X.4.3 Detection of read / write violations in I/O registers

The *access_type* specification can be extended with an extra parameter, which makes it possible to detect illegal use of an I/O register at compile time.

The minimal parameter set for a read / write limitation specification would be:

- **S** Defined as Read-only register
- **S** Defined as Write-only register
- **S** Defined as Read-modify-Write register (behaves like a RAM cell)

**Table: Allowed operations on different I/O register types:**

| | **iowrxx** | **iordxx** | **ioorxx** | **ioandxx** | **ioxorxx** |
|---|---|---|---|---|---|
| **Read-Write** | *Yes* | *Yes* | *Yes* | *Yes* | *Yes* |
| **Write-only** | *Yes* | *No* | *No* | *No* | *No* |
| **Read-only** | *No* | *Yes* | *No* | *No* | *No* |

The not-allowed cases should generate some kind of error message at compile time. With a template implementation of *iohw,* the compiler will usually at least complain that no matching template function can be found for the not-allowed cases.

**Example:**

```
// Define type to validate I/O register access
enum rw_t                 // Access mode type
  {
  read,                   // Read only access
  write,                  // Write only access
  read_write              // Read, Write or Read-Modify-Write access
  };

// Define access_type template for direct addressing
template <class T, address_t address,
  bus_t buswidth, chip_t chiptype, rw_t access>
    class IO_MM { };

//--- User declaration of I/O registers in platform
//--- (normally placed in a separate platform dependent header file)
typedef IO_MM <uint8_t, 10800, bw8, chip8, write>      WR_PORT;
typedef IO_MM <uint8_t, 20800, bw8, chip8, read>       RD_PORT;
typedef IO_MM <uint8_t, 30800, bw8, chip8, read_write> RDWR_PORT;

// User code
uint8_t myval;
myval = iord8(RD_PORT);          // ok
myval += iord8(RDWR_PORT);       // ok
iowr8(WR_PORT,myval);            // ok
iowr8(RDWR_PORT,0x45);           // ok

myval = iord8(WR_PORT);          // Illegal, generate compile time error
iowr8(RD_PORT,0x55);             // Illegal, generate compile time error
```

## X.5 Other implementation considerations

### X.5.1 Atomic operation

It is an *iohw* implementation requirement that in each I/O function a given (partial) I/O register is addressed exactly once during a read or a write operation and exactly twice during a read-modify-write operation.

It is an *iohw* implementation recommendation that each I/O function be implemented so that the I/O access operation becomes *atomic* whenever possible.

However, atomic operation is not guaranteed to be portable across platforms for read-modify-write operations (*ioorxx, ioandxx, ioxorxx*) or for multi-addressing cases.

The reason for this is simply that many processor architectures do not have the instruction set features required for assuring atomic operation.

## X.5.2 Read-modify-write operations in multi-addressing cases.

Read-modify-write operations should, in general, do a complete read of the I/O register, followed by the operation, followed by a complete write to the I/O register.

It is therefore recommended that an implementation of multi-addressing cases *should not* use read-modify-write machine instructions during *partial* register addressing operations.

The rationale for this restriction is to use the lowest common denominator of multi-addressing hardware implementations in order to support as wide a range of I/O hardware register implementation as possible.
For instance, more advanced multi-addressing I/O register implementations often take a snap-shot of the whole logical I/O register when the first partial register is being read, so that data will be stable and consistent during the whole read operation. Similarly, write registers are often made double-buffered so that a consistent data set is presented to the internal logic at the time when the access operation is completed by the last partial write.
Such hardware implementations often require that each access operation is completed before the next access operation is initiated.

## X.6 Typical implementation optimization possibilities

*Pre-calculation of constant expressions*
All constant expressions should be solved at compile time. Using *inline* functions, both interleave factors and constant buffer indexes should therefore be folded into the address value(s) used in the machine code.

Therefore the following two I/O write statements should result in exactly the same machine code:

```
iowr8(PORT1,0x33);
iowrbuf8(PORT1, 0, 0x33);
```

An implementation can take advantage of this, because the number of I/O functions which have to be implemented can be reduced with no efficiency penalty using simple macro definitions like:

```
#define iowr8(access_type,val)  iowrbuf8(access_type,0,(val))
```

*Multi-addressing and endian*
Typical candidates for platform dependent optimization are I/O functions for the multi-addressing cases (logical I/O register width > I/O chip bus width) where the width of the chip data bus matches the width of the processor data bus. In these cases, multi-byte access can often use data types directly supported by the processor for either the lsb or msb endian functions. The other endian functions can often be implemented efficiently using one load or store operation plus one or more register swap operations.