J16/01-0004 = WG21 N1290 (= WG14 N936 Draft **ISO/IEC WDTR 18037)**

WG14 Technical Report
Draft for the pre-Copenhagen mailing of

Extensions for the programming language C to support embedded processors

For use by the C++ performance group.

Chapter 3 in this document describes the new interface for Basic I/O hardware addressing as it is seen from C.

The interface description has now been updated to reflect the modifications agreed upon at the Toronto meeting. The major change is that the interface is made data type independent. The data type for an I/O access operation is now defined by the access_type type instead of by the function parameters.

Annex B describe implementation considerations. (It is mostly a collection of the language independent parts of N1233).

To do: The goal is that it become possible to use the same interface from C and C++. Text in the performance group TR should therefore be updated to describe how the same interface for basic I/O hardware addressing can be implemented based on C++ templates.

At the Copenhagen meeting we must also decide in principle: Should the WG21 TR contain a copy of the interface description and annex B, or should the WG21 TR just make a (normative) reference to the WG14 TR?

Jan Kristoffersen
Member of Danish Standard

J16/01-0004 = WG21 N1290 (= WG14 N936 Draft **ISO/IEC WDTR 18037)**

WG14 N936

ISO/IEC WDTR 18037.1

Programming languages, their environments and system software interfaces — Extensions for the programming language C to support embedded processors

Contents

INTRODUCTION

In the fast growing market of embedded systems there is an increasing need to write application programs in a high-level language such as C. Basically there are two reasons for this trend: programs for embedded systems get more complex (and hence are difficult to maintain in assembly language) and the different types of embedded systems processors have a decreasing lifespan (which implies more frequent re-adapting of the applications to the new instruction set). The code re-usability achieved by C-level programming is considered to be a major step forward in addressing these issues.

Various technical areas have been identified where functionality offered by processors (such as DSPs) that are used in embedded systems cannot easily be exploited by applications written in C. Examples are fixed-point operations, usage of different memory spaces, low level I/O operations and others. The current proposal addresses only a few of these technical areas.

Embedded processors are often used to analyse analogue signals and process these signals by applying filtering algorithms to the data received. Typical applications can be found in all wireless devices. The common datatype used in filtering algorithms is the fixed point datatype, and in order to achieve the necessary speed, the embedded processors are often equipped with special hardware support that datatype. Standard C (as defined in ISO/IEC 9899:1999) does not provide support the fixed point arithmetic operations, currently leaving programmers with no option but to hand-craft most of their algorithms in assembler. This Technical Report specifies a fixed point datatype for C, definable in a range of precision and saturation options. In this manner, fixed point data is supported as easily as integer and floating point data throughout the compiler, including the critical optimisers leading to highly efficient code.

Typical for the mentioned filtering algorithms is the usage of polynomials whereby data from one source (inputvalues) is multiplied by coefficients coming from another source (memory). Ensuring the simultaneous flow of data and coefficient data to the multiplier/accumulator of processors designed for FIR filtering, for example, is critical to their operation. In order to allow the programmer to declare the memory space from which a specific data object must be fetched. This Technical Report specifies support for multiple memory spaces of dual Harvard architectures DSP processors. As a result, optimising compilers can utilise the ability of these processors to read data from two separate memories in a single cycle to maximise execution speed.

Another feature of many DSP algorithms, including FIR filters and FFTs, is that they frequently use the same block of data over and over again, working from the start of the block to the end and then looping back to the beginning. Although the optimum way of achieving this is to create a circular buffer, such buffers are not easily defined in standard C programs. This Technical Report therefore includes a language extension that allows the size and memory space of circular buffers to be defined so that they can be easily generated during program compilation.

[Editor's note: Are all above paragraphs necessary?]

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.

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.

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

Information technology — Programming languages, their environments and system software interfaces — Extensions for the programming language C to support embedded processors

1 General

1.1 Scope

This Technical Report specifies a series of extensions of the programming language C, specified by the international standard ISO/IEC 9899:1999.

Each clause in this Technical Report deals with a specific topic. The first subclause of each clause contains a technical description of the features of the topic. It provides an overview but does not contain all the fine details. The second subclause of each clause contains the editorial changes to the standard, necessary to fully specify the topic in the standard, and thereby provides a complete definition. .If necessary, additional explanation and/or rationale is given in an Annex.

1.2 References

The following standards contain provisions which, through reference in this text, constitute provisions of Technical Report. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this Technical Report are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred applies. Members of IEC and ISO maintain registers of current valid International Standards.

ISO/IEC 9899:1999, *Information technology – Programming languages, their environments and system software interfaces – Programming Language C.*

2 Fixed point arithmetic

2.1 Overview and principles of the fixed point datatype

2.1.1 The datatypes

Fixed point datavalues are either integer datavalues, fractional datavalues (with value between -1.0 and +1.0), or datavalues with an integral part and a fractional part. As the position of the radix point is known implicitly, operations on the values of these datatypes can be implemented with (almost) the same efficiency as operations on integral values. Typical usage of fixed point datavalues and

operations can be found in applications that convert analogue values to digital representations and subsequently apply some filtering algorithm. For more information of fixed point datatypes, see clause A.1 in the Annex of this Technical Report.

For the purpose of this Technical Report, two new datatypes are added to the C language: the **`fixed`** datatype and the **`accum`** datatype. The (optionally **`unsigned`**) **`fixed`** datatype has no integral part, hence values of the **`fixed`** datatype are between -1.0 and +1.0. The value of an **`accum`** datavalue depends on the number of integral bits in the datatype. Note that the **`fixed`** datatype corresponds with the type-A datatype, as described in the Annex, while the **`accum`** datatype corresponds to the type-B datatype, mentioned in the Annex.

The fixed point datatypes are designated with the corresponding new keywords and *type-specifier*s **`fixed`** and **`accum`**. These *type-specifier*s can be used in combination with the existing *type-specifier*s **`short`**, **`int`**, **`long`**, **`signed`** and **`unsigned`** to designate the allowed fixed point types, yielding

```
short fixed          short accum
int fixed            int accum
fixed                accum
long fixed           long accum
long long fixed      long long accum
```

and their signed and unsigned variations.

The following interpretations and/or restrictions apply:

1. The **`int fixed`** (or **`int accum`**) type is the same type as the **`fixed`** (or **`accum`**) type.
2. If neither of the **`signed`** or **`unsigned`** keywords is used, a signed fixed point datatype is implied.
3. It is implementation defined whether unsigned fixed point datatypes are supported.
4. The number of integral bits and fractional bits in a fixed point datatype is implementation defined.
5. A conforming implementation shall support at least two different signed **`fixed`** fixed point datatypes, and one signed **`accum`** fixed point datatype.
6. Every fixed point datatype has a *fixed point conversion rank*, which is defined as follows:
   - the rank of **`long long fixed`** shall be greater than the rank of **`long fixed`**, which shall be greater than the rank of **`fixed`,** which shall be greater than the rank of **`short fixed`**;
   - the rank of **`long long accum`** shall be greater than the rank of **`long accum`**, which shall be greater than the rank of **`accum`,** which shall be greater than the rank of **`short accum;`**
   - the rank of **`long long accum`** shall be greater than the rank of **`long long fixed`**, the rank of **`long accum`** shall be greater than the rank of **`long fixed`**, the rank of **`accum`** shall be greater than the rank of **`fixed`**, the rank of **`short accum`** shall be greater than the rank of **`short fixed`**.
7. For each two different fixed point datatypes, the type with the higher rank shall have at least the same number of integral bits and fractional bits as the other type.

8. The concept *container* is used to identify the address and the size (expressed in bytes) of a fixed point datavalue. A container is composed of a contiguous sequence of one or more bytes, holding a fixed point datavalue. Some requirements:
   - the size of the container in bits is a multiple of the number of bits per byte for the machine, and the address of the container is a regular byte address;
   - the number of databits (fractional bits and integral bits) in a fixed point datavalue is not greater than the number of bits in its container;
   - the (machine) address of a fixed point datavalue is the (machine) address of (exactly) one of the bytes that form the container;
   - if the size of the container in bits is greater than the number of bits needed for the fixed point datavalue, the remaining bits (called *paddingbits*) cannot be used for other purposes;
   - if there are paddingbits involved, it is still required that the (machine) address of (one of the bytes of) the container fully identifies the (machine) address of the fixed point datavalue; in other words: the alignment of the fixed point datavalue within the container is implicitly known (from its fixed point datatype designation);
   - at programming level (i.e., in the programming language) all fixed point datatypes with the same valuespace (the same number of databits, same signedness, same position of the radix point) are the same; there is no distinction between these datatypes with respect to different alignment/padding strategies.

2.1.2 Saturation

Saturation is a mode associated with a type. When a value is converted to a saturated type and that value is too large (or too small) to be represented by the type, the maximal (or minimal) value (according to the type) is assigned instead. In other circumstances an overflow condition would have been created.

For the purpose of this Technical Report, a new keyword **`sat`** is introduced. This keyword can be used, in combination with the **`fixed`** and **`accum`** type specifiers, to define saturated fixed point datatypes.

A saturated fixed point datatype has the same fixed point conversion rank as the corresponding non-saturated fixed point datatype.

Note: the **`sat`** keyword can also be used to define saturated arithmetic types in general. This Technical Report will not elaborate on this usage.

2.1.3 Modular wraparound

Some systems support modular wraparound on overflow; this can be specified in the same way as saturation (with a **`mod`** keyword?). **TEXT TO BE ADDED.**

2.1.4 Fixed point operators

The unary operators **+** and **-**, and the binary operators **+**, **-**, ***** and **/** are supported for fixed point datatypes.

For binary operators, the following conversion rules shall apply:
- if the types of the operands have the same fixed point conversion rank, then the result type shall have that same fixed point conversion rank;
- otherwise, the operand with the type with the smaller fixed point conversion rank shall be converted to the other type, and the result type shall be the latter type;
- if either of the operands as a saturated type, then the result type shall be saturated.

2.1.5 Usual arithmetic conversions, type casts

See discussion text in Annex.

2.1.6 Fixed point constants

See discussion text in Annex.

2.2 Detailed changes to ISO/IEC 9899:1999

3 Basic I/O hardware addressing < iohw.h header>

The purpose with the I/O hardware access functions defined in a new header file <iohw.h> is to promote portability of I/O hardware driver source code across different execution environments.

3.1 Overview and principles

The new I/O hardware access functions create a simple and platform independent interface between I/O driver source code and the underlying access methods used when addressing the I/O registers in a given platform.

The primary purpose with the interface is to separate characteristics which are portable and specific for a given I/O register, for instance the register bit width, from characteristics which are related to a specific execution environment, for instance the I/O register address, the processor bus type and endian, chip bus size and endian, address interleave, the compiler access method etc. Use of this separation principle enables I/O driver source code itself to be portable to all platforms where the I/O registers can be connected.

An I/O register must always be referred in the driver source code with a symbolic name. The symbolic name must refer to a complete definition of the access method used with the given register. The I/O syntax standardisation method creates a conceptually simple model for I/O registers:

Symbolic name for I/O register <-> Complete definition of the access method

When porting the I/O driver source code to a new platform then only the definition of the access method (definition of the symbolic name) need to be updated.

Example:

It is convenient to locate all I/O register name definitions in a separate header file (called iohw_ta.h in the following). A typical I/O driver setup then operates with minimum three C modules.

I/O driver module	The I/O driver C source code. Portable across compilers and platforms. Includes IOHW.H and IOHW_TA.H
IOHW.H	Defines I/O functions and access methods Typically specific for a given compiler. Implemented by the compiler vendor.
IOHW_TA.H	Defines symbolic I/O register names and the corresponding access methods. Specific for the given execution environment. Implemented and maintained by the programmer.

Example:

```
#include <iohw.h>
#include <iohw_ta.h>   // my I/O register definitions for target

unsigned char mybuf[10];
//..
iowr(MYPORT1, 0x8);                 // write single register
for (int i = 0; i < 10; i++)
    mybuf[i] = iordbuf(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.

3.1.1 I/O register characteristics

The principle behind the iohw interface is that all I/O register characteristics should be visible from the driver source code while all platform specific characteristics are encapsulated by the header files and the underlying iohw implementation.

I/O registers often behave differently from the traditional memory model. They may be read-only, write-only or read-modify-write types, often read and write operations are only allowed once for each event, etc.
All such I/O register specific characteristic should be visible at the I/O driver code level and should not be hidden by the iohw interface implementation.

3.1.2 The most basic I/O operations

The most basic operations on I/O register hardware are READ and WRITE.
Bit set, bit-clear and bit-invert of individual bits in an I/O hardware register are commonly used operations. Many processors have special machine instructions for doing this.
For the convenience of the programmers, and in order to promote good compiler optimization for bit operations, the basic logical operations OR, AND and XOR are defined by the iohw interface in addition to READ and WRITE.

All other arithmetic and logical operations used by the driver source code can be build on top of these few basic I/O operations.

3.1.3 The access_type

The *access_types* defined in the header <iohw.h> are a new kind of types which are only intended to be used as parameters in the macro functions for I/O register access.

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 definition method and the implementation of *access_types* are processor and platform specific.

In general an access_type definition will specify at least the following characteristics:

Register size (mapping to a C data type).
Access limitations (read-only, write-only)
Bus address for register.

Other access characteristics typically specified via the access_type:

Processor bus (if more than one).
Access method (if more than one).
I/O register endian (if register width is larger than the chip bus width)
Interleave factor for I/O register buffers (if bus width for the chip is smaller)
User supplied access driver functions.

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.

Further details and implementation considerations are discussed in annex **B**, **C** and **D**.

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.

3.2 The IOHW interface

The header <iohw.h> declare several function macros which together creates a data type independent interface for basic I/O hardware addressing.

3.2.1 Functions for single register access

Synopsis

```
#include <iohw.h>
iord(  access_type )
iowr(  access_type, value )
ioor(  access_type, value )
ioand( access_type, value )
ioxor( access_type, value )
```

Description

These macros maps a I/O hardware register operation to an underlying (platform specific) implementation which provide access to the I/O register identified by *access_type,* and perform the basic operation *rd*, *wr*, *or*, *and* or *xor* as identified by the function name on this register.

The data type (the I/O register size) for *value* parameters and the value returned by *iord()* is defined by the *access_type* definition for the given register. The macro functions *iowr, ioor, ioand* and *ioxor* do not return a value.

3.2.2 Functions for register buffer access

Synopsis

```
#include <iohw.h>
iordbuf(  access_type, index )
iowrbuf(  access_type, index, value )
ioorbuf(  access_type, index, value )
ioandbuf( access_type, index, value )
ioxorbuf( access_type, index, value )
```

Description
These macros maps a I/O hardware register buffer operation to an underlying (platform specific) implementation which provide access to the I/O register buffer identified by *access_type,* and perform the basic operation *rd*, *wr*, *or*, *and* or *xor* as identified by the function name on this register.

The data type (the I/O register size) for *value* parameters and the value returned by *iordbuf()* is defined by the *access_type* definition for the given register. The macro functions *iowrbuf, ioorbuf, ioandbuf* and *ioxorbuf* do not return a value.

The *index* parameter is offset in the register 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 address than element n.

It should be noted that the *index* parameter is the offset in the I/O hardware buffer, not the processor address offset. Conversion from a logical index to a physical address require that *interleave* calculations are performed by the underlying implementation. This is described in further details in **B.2.4**

3.2.3 Function for access_type initialization

Synopsis

```
#include <iohw.h>
io_at_init( access_type )
io_at_release( access_type )
```

Description

The *io_at_init* macro maps to an underlying (platform specific) implementation which provide any access_type initialization necessary *before* performing any other operation on the I/O register (set of I/O registers) identified by *access_type*. This macro should be placed in the driver source code so it is invoked at least once before any other operations on the related registers are performed. The macro do not return a value.

The *io_at_release* macro maps to an underlying (platform specific) implementation which release any resources obtained by a previous call to *io_at_init* for the same *access_type*. This macro should be placed in the driver source code so it is invoked once after all operations on the related registers have stopped. The macro do not return a value.

Example:

With an implementation for a hosted environment the *io_at_init* macro can be used for identifying the point in an execution sequence where the underlying access method should obtain, or have obtained, a handler from the operating system. *io_at_exit* can be used to identify the point in an execution sequence where the handler can returned to the operating system.

If no *access_type* initialization is required by a given <iohw.h> header implementation the *io_at_init* and *io_at_release* macro definition may be empty.

3.2.4 Function for access_type copying

Synopsis

```
#include <iohw.h>
io_at_cpy( access_type_dest, access_type_src)
```

Description

The macro maps to an underlying (platform specific) implementation which copy the dynamic part of the source *access_type* to the destination *access_type*. The two parameters is assumed to be of the same *access_type* type. The macro do not return a value.

If no *access_type* copying is supported by a given <iohw.h> header implementation or a given *access_type* type does not contain any dynamic elements the *io_at_cpy* macro may result in an empty statement.

A typical use for *io_at_cpy* is when a set of driver functions for a given I/O chip type are used with multiple hardware instances of the same chip. It often provides a faster alternative to passing the *access_type* as a function parameter.

Example

```
#include <iohw.h>
#include <iohw_ta.h>  // MYCHIP_CFG and MYCHIP_DATA are defined
                      // relative to a dynamic MYCHIP_BASE
```

```
// Portable driver function
uint8_t my_chip_driver(void)
{
    iowr(MYCHIP_CFG, 0x33);
    return iord(MYCHIP_DATA);
}

// Users driver application
uint8_t  d1,d2;
// Read from our 2 I/O chips
io_art_cpy(MYCHIP_BASE, CHIP1); // Select chip 1
d1 = my_chip_driver();
io_art_cpy(MYCHIP_BASE, CHIP2); // Select chip 2
d2 = my_chip_driver();
```

Annex A

Additional information and Rationale

A.1 Fixed point

A.1.1 Fixed point types

The set of representable floating point values (which is a subset of the real values) is characterized by a sign, a precision and the position of the radix point. For those values that are commonly denoted as floating point values, the characterizing parameters are defined within a format (such as the IEEE formats or the VAX floating point formats), usually supported by hardware instructions, that defines the size of the container, the size (and position within the container) of the exponent, and the size (and position within the container) of the sign. The remaining part of the container then contains the mantissa. [The formats discussed in this section are assumed to be binary floating point formats, with sizes expressed in bits. A generalization to other radixes (like radix-10) is possible, but not done here.] The value of the exponent then defines the position of the radix point. Common hardware support for floating point operations implements a limited number of floating point formats, usually characterized by the size of the container (32-bits, 64-bits etc); within the container the number of bits allocated for the exponent (and thus for the mantissa) is fixed. For programming languages this leads to a small number of distinct floating point datatypes (for C these are **`float`**, **`double`**, and **`long double`**), each with its own set of representable values.

For fixed point types, the story is slightly more complicated: a fixed point value is characterized by its precision (the number of databits in the fixed point value) and an optional signbit, while the position of the radix point is defined implicitly (i.e., outside the format representation): it is not possible to deduct the position of the radix point within a fixed point datavalue (and hence the value of that fixed point datavalue!) by simply looking at the representation of that datavalue. It is however clear that, for proper interpretation of the values, the hardware (or software) implementing the operations on the fixed point values should know where the radix point is positioned. From a theoretical point of view this leads (for each number of databits in a fixed point datatype) to an infinite number of different fixed point datatypes (the radix point can be located anywhere before, in or after the bits comprising the value).
There is no (known) hardware available that can implement all possible fixed point datatypes, and, unfortunately, each hardware manufacturer has made its own selection, depending on the field of application of the processor implementing the fixed point datatype. Unless a complete dynamic or a parameterized typesystem is used (not part of the current C standard, hence not proposed here), for programming language support of fixed point datatypes a number of choices need to be made to limit the number of allowable (and/or supported or to be supported) fixed point datatypes. In order to give some guidance for those choices, some aspects of fixed point datavalues and their uses are investigated here.

For the sake of this discussion, a fixed point datavalue is assumed to consist of a number of databits and a signbit. On some systems, the signbit can be used as an extra databit, thereby creating an unsigned fixed point datatype with a larger (positive) maximum value.
Note that the size of (the number of bits used for) a fixed point datavalue does not necessarily equal the size of the container in which the fixed point datavalue is contained (or through which the fixed point datavalue is addressed): there may be gaps here!

As stated before, it is necessary, when using a fixed point datavalue, to know the place of the radix point. There are several possibilities.
The radix point is located immediately to the right of the rightmost (least significant) bit of the databits. This is a form of the ordinary integer datatype, and does not (for this discussion) form part of the fixed point datatypes.

- The radix point is located further to the right of the rightmost (least significant) bit of the databits. This is a form of an integer datatype (for large, but not very precise integer values) that is normally not supported by (fixed point) hardware. In this document, these fixed point datatypes will not be taken into account.
- The radix point is located to the left of (but not adjacent to) the leftmost (most significant) bit of the databits. It is not clear whether this category should be taken into account: when the radix point is only a few bits away, it could be more 'natural' to use a datatype with more bits; in any case this datatype can easily (??) be simulated by using appropriate normalize (shift left/right) operations. There is no known fixed point hardware that supports this datatype.
- The radix point is located immediately to the left of the leftmost (most significant) bit of the databits. This datatype has values (for signed datatypes) in the interval (-1,+1), or (for unsigned datatypes) in the interval [0,1). This is a very common, hardware supported, fixed point datatype. In the rest of this section, this fixed point datatype will be called the type-A fixed point datatype. Note that for each number of databits, there are one (signed) or two (signed and unsigned) possible type-A fixed point datatypes.
- The radix point is located somewhere between the leftmost and the rightmost bit of the databits. The datavalues for this fixed point datatype (type-B fixed point datatypes) have an integral part and a fractional part. Some of these fixed point datatypes are regularly supported by hardware. For each number of databits N, there are (N-1) (signed) or (2*N-1) (signed and unsigned) possible type-B fixed point datatypes.

Apart from the position of the radix point, there are three more aspects that influence the amount of possible fixed point datatypes: the presence of a signbit, the number of databits comprising the fixed point datavalues and the size of the container in which the fixed point datavalues are stored.
In the embedded processor world, support for unsigned fixed point datatypes is rare; normally only signed fixed point datatypes are supported. However, to disallow signed fixed point arithmetic from programming languages (in general, and from C in particular) based on this observation, seems overly restrictive.

There are two further design criteria that should be considered when defining the nature of the fixed point datatypes:

- it should be possible to generate optimal fixed point code for various processors, supporting different sized fixed point datatypes (examples could include an 8-bit fixed point datatype, but also a 6-bit fixed point datatype in an 8-bit container, or a 12-bit fixed point datatype in a 16-bit container);

- it should be possible to write fixed point algorithms that are independent of the actual fixed point hardware support. This implies that a programmer (or a running program) should have access to all parameters that define the behaviour of the underlying hardware (in other words: even if these parameters are implementation defined).

With the above observations in mind, the following recommendations are made.

1. Introduce **`signed`** and **`unsigned`** fixed point datatypes, and use the existing signed and unsigned keywords (in the 'normal' C-fashion) to distinguish these types. Omission of either keywords implies a signed fixed point datatype.
2. Introduce a new keyword and *type-specifier* **`fixed`** (similar to the existing keyword **`int`**), and define the following five standard signed fixed point types: **`char fixed`**, **`short fixed`**, **`fixed`**, **`long fixed`** and **`long long fixed`**. The supported (or required) underlying fixed point datatypes are mapped on the above in an implementation-defined manner, but in a non-decreasing order with respect to the number of databits in the corresponding fixed point datavalue. Note that there is not necessarily a correspondence between a fixed point datatype designator and the type of its container: when an 18-bit and a 30-bit fixed point datatype are supported, the 18-bit will probably have the **`short fixed`** type and the 30-bit type will probably have the **`fixed`** type, while the containers of these types will be the same.
3. If more fixed point datatypes are needed, (or if there is a need to better distinguish certain fixed point datatypes), an approach similar to the **`<stdint.h>`** approach could be taken, whereby **`fixed_le`***`N`*`_t`** could designate a (type-A) fixed point datatype with at least N databits, while **`fixed_le`***`M`*`_le`***`N`*`_t`** could designate a (type-B) fixed point datatype with at least M integral bits and N fractional bits.
4. In order for the programmer to be able to write portable algorithms using fixed point datatypes, information on (and/or control over) the nature and precision of the underlying fixed point datatypes should be provided. The normal C-way of doing this is by defining macro names (like **`SHORT_FIXED_FRAC_BITS`** etc.) that should be defined in an implementation defined manner.

A.1.2 Usual arithmetic conversions, type casts

It is proposed to situate the fixed point datatypes 'between' the integer datatypes and the floating point datatypes: if only integer datatypes are involved then the current standard rules (cf. 6.3.1.1 and 6.3.1.8) are followed, when fixed point operands but no floating point operands are involved the operation will be done using fixed point datatypes, otherwise everything will be converted to the appropriate floating point datatype.

Since it is likely that an implementation will support more than one (type-A and/or type-B) fixed point datatype, in order to assure arithmetic consistency it should be well-defined to which fixed point datatype a type is converted to before an operation involving fixed point and integer datatvalues is performed. There are several approaches that could be followed here:

- define that the result of any operation on fixed point datatypes should be as if the operation is done using infinite precision. This gives an implementation the possibility to choose an implementation dependent optimal way of calculating the result (depending on the required

precision of the expression by selecting certain fixed point operations, or, maybe, emulate the fixed point expression in a floating point unit), as long as the required result is obtained.
- to define a (implementation defined?) extended fixed point datatype to which every operand is converted before the operation. It is then important that the programmer has access to the parameters of this extended fixed point type in order to control the arithmetic and its results. This could either be the 'largest' type-B fixed point datatype (if supported), or the 'largest' type-A fixed point datatype.

A.1.3 Fixed point constants

There are currently two approaches towards fixed point constants:

1. in the `normal' C fashion, by appending a suffix (or a combination of suffixes) to the string denoting the value of the constant (much like section 6.4.4 of the C standard); or
2. with special syntax "<type-name> (<constant-expression>)".

Both approaches have pro's and con's. It needs to be discussed which approach should be used.

Annex B

Implementing the iohw header

(Informative annex)

B.1 General

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

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.

B.1.1 Recommended steps

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

1. 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.

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

3. 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.

B.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 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.

Simple *iohw* implementations limited to the most basic functionality can be implemented efficiently using a mixture of macros, *in-line* functions and intrinsic types or functions. See Annex D regarding simple macro implementations.

Full featured implementations of iohw will require direct compiler support for access_types. See Annex C regarding a generic access_type descriptor.

B.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.

B.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)

B.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)

B.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 for the specified bus widths.

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)

B.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 *logical_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*.

B.3 Access_types for different I/O addressing methods

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

- *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.

- *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.

- *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.

- *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

B.4 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 (*ioor, ioand, ioxor*) 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.

B.5 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.

B.6 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 the 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.

A standard 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.

Another solution is to use *access_type* copying and *access_types* with dynamic access information. *io_at_cpy(access_type_dest, access_type_src)* provides a portable way to do this.

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.

io_at_init(access_type) can be used as a portable way to specify in the source code where and when such initialization should take place.

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 an 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.

Annex C

Generic access_type descriptor for I/O hardware addressing

(Informative annex)

C.1 Generic access_type descriptor

This informative annex proposes a consistent and complete specification syntax for defining I/O registers and their access methods in C.

C.1.1 Background

Current work has shown that there are three basic requirements which must not be compromised by any standardized solution for portable I/O register access:

- The symbolic I/O register name used in the I/O driver code must refer to a **complete definition of the access method**.
- The standardized solution must be able to **encapsulate** all knowledge about the underlying processor, platform, and bus system.
- It should provide a **no-overhead solution** (for simple access methods).

In order to fulfill the first two requirements in a consistent way, it should be possible *to refer to a complete access_type specification as a single entity*. This is necessary, for instance, to pass *access_type* parameters between functions.

This can been achieved in several different ways. Prior art has used a number of (intrinsic) memory type qualifiers or special keywords, which have varied from compiler to compiler and from platform to platform.

However, type qualifiers have always tended to be an inadequate description method when more complex access methods are needed. For instance, it must be possible to encapsulate all access method variation possible in the target platform. These differences include the widths of I/O registers, and the qualities of the I/O chip bus and processor bus: register interleave values, I/O register endian specifications, and so on. Similarly, type qualifiers are usually inadequate when more complex addressing methods are used (base pointer addressing, pseudo-bus addressing, addressing via user device drivers, and others).

This paper proposes a generic syntax for defining the *access_type* for an I/O register. The syntax is a new approach and a super-set solution, intended to replace prior art.

C.2 Syntax specification

Access_type specification:

typedef ACCESS_METHOD_NAME < parameter list > SYMBOLIC_PORT_NAME;

parameter list:
access method independent parameter list , access method specific parameter list

access method independent parameter list:
type for I/O register value (size of I/O register) ,
access limitation type ,
I/O register chip bus type (size and endian of I/O chip bus)

type for I/O register value (size of I/O register):
uint8_t
uint16_t
uint32_t
uint64_t
bool
(+ optionally any basic type native to the implementation)

access limitation type: // for compile time diagnostic
ro_t *//read_only*
wo_t *//write_only*
rw_t *//read_write*
rmw_t *//read_modify_write*

I/O register chip bus type:
chip8 *// register width = chip bus width = 8 bit*
chip8l *// register width > chip bus width, MSB on low address*
chip8h *// register width > chip bus width, MSB on high address*
chip16 *// register width = chip bus width = 16 bit*
chip16l *// register width > chip bus width, MSB on low address*
chip16h *// register width > chip bus width, MSB on high address*
chip32 *// register width = chip bus width = 32 bit*
chip32l *// register width > chip bus width, MSB on low address*
chip32h *// register width > chip bus width, MSB on high address*
chip64 *// register width = chip bus width = 64 bit*
(+ optionally any bus width native to the implementation)

access method specific parameter list:
// Depends on the given access method. Examples are given later.
// Three typical parameters are:
Primary address constant ,
Processor bus width type,
Address mask constant

Processor bus width type:
bw8 *// 8 bit bus*
bw16 *// 16 bit bus*
bw32 *// 32 bit bus*
bw64 *// 64 bit bus*
(I.e. any bus widths native to the implementation)

Angled brackets are employed as delimiters for the access type parameter list, to distinguish it from a function parameter list (using "()") and from a structure or enumerated list (using "{}").

An implementation must define at least one access method for each processor addressing range. For instance, for the 80x86 CPU family, an implementation must define at least two access_methods, one for the memory-mapped range, and one for the I/O-mapped range.
If several different access methods are supported for a given address range, then an access type must exist for each access method.

The *ACCESS_METHOD_NAME* is an identifier for the parameter set enclosed in angled brackets. It is an implementation-defined keyword which tells the compiler how to interpret the parameter set. A compiler will typically support a number of different *access_type* descriptors.

C.3 Examples of access_type descriptors

Below are some examples of *access_type* parameter combinations for different (typical) access methods:

Direct addressing:

```
typedef MM_DIRECT <
        type for I/O register value (size of I/O register),
        access limitation type,
        I/O register chip bus type (size and endian of I/O chip bus),
        primary address constant,
        processor bus width type
        > PORT_NAME;
```

The I/O register at the primary address is addressed directly. If the bit width of the I/O register is larger than the I/O chip bus width, then the access operation is built from multiple consecutive addressing operations.

Based addressing:

```
typedef MM_BASED <
        type for I/O register value (size of I/O register),
        access limitation type,
        I/O register chip bus type (size and endian of I/O chip bus),
        primary address constant,
        processor bus width type,
        base variable
        > PORT_NAME;
```

The I/O register at the *primary_address* + value of *base_variable* is addressed directly. If the bit width of the I/O register is larger than the I/O chip bus width, then the access operation is built from multiple consecutive addressing operations.

Indexed-bus addressing:

```
typedef MM_INDEXED <
        type for I/O register value (size of I/O register) ,
        access limitation type ,
        I/O register chip bus type (size and endian of I/O chip bus),
```

```
        primary address constant,
        processor bus width type,
        secondary address parameter
        > PORT_NAME;
```

The I/O register on an indexed bus (also called a pseudo-bus) is addressed in the following way. The primary address is written to the register given by the secondary address parameter (= initiate indexed bus address). The access operation itself is then done on the location (secondary address parameter+1 = data at indexed bus).
This method is a common way to save addressing bandwidth. The method also makes it particularly easy to connect chips using a multiplexed address/data bus interface to a processor system having a non-multiplexed interface.

Device driver addressing:

```
typedef MM_DEVICE_DRIVER <
        type for I/O register value (size of I/O register) ,
        access limitation type ,
        I/O register chip bus type (size and endian of I/O chip bus),
        primary address constant,
        processor bus width type,
        name of driver function for register write,
        name of driver function for register read
        > PORT_NAME;
```

The I/O register is addressed by invoking (user-defined) driver functions. If the bit width of the I/O register is larger than the I/O chip bus width, then the access operation is built from multiple consecutive addressing operations. (Alternatively, the I/O register chip bus type, processor bus width type and the primary address could be transferred to the driver functions.)

Direct bit addressing:

```
typedef MM_BIT_DIRECT <
        type for I/O register value (size of I/O register),
        access limitation type,
        I/O register chip bus type (size and endian of I/O chip bus),
        primary address constant,
        processor bus width type,
        bit location in register constant,
        > PORT_NAME;
```

The I/O register at the primary address is addressed directly.

Examples:

```
typedef MM_DIRECT <uint8_t,rw_t,chip8,0x3000,bw8> MYPORT;
uint8_t a = iord(MYPORT,0xAA); // Read single register
```

MYPORT is an 8-bit read-write register, located in a chip with an 8-bit data bus, connected to a (memory-mapped) 8-bit processor bus at address 0x3000.

```
typedef MM_DIRECT <uint16_t,wo_t,chip8l,0x200,bw16> PORTA;
iowr(PORTA,0xAA); // Write single register
```

PORTA is a 16-bit write-only register, located in a chip with an 8-bit data bus (with MSB register part located at the lowest address), where the chip is connected to a (memory-mapped) 16-bit processor bus at address 0x200.

Use of user-defined device drivers:

```
// Memory buffer addressed via user-defined access drivers
typedef MM_DEVICE_DRIVER
       <uint8_t,rmw_t,chip8,0xA,my_wr_drv,my_rd_drv> DRVREG;

// User-defined read driver to be invoked by compiler
inline uint8_t void my_rd_drv( int index )
       {
       // some driver code
       }

// User-defined wr driver to be invoked by compiler
inline void my_wr_drv( int index , uint8_t dat)
       {
       // some driver code
       }

// user code
int i;
i = iord(DRVREG);          // = call of my_rd_drv(0xA);
for (i = 0; i < 0xA0; i++)
    iowrbuf(DRVREG,i,0x0); // = call of my_wr_drv(i+0xA,0)
```

C.4 Parsing

The access type descriptors are parsed at compile time.

If the symbolic port name is used directly in iord(..)/iowr(..)/etc. functions, the code can be completely optimized at compile time: all information for doing this is available to the compiler at that stage. Based on the combined parameter set (the types), the compiler will typically select among several internal intrinsic inline access functions to generate the appropriate code for the access operation. No memory instantiation of an *access_type* object is needed. This will fulfill the third of the primary requirements on page 1 (no-overhead solution).

Example:

```
typedef MM_DIRECT<uint16_t,rmw,chip8l,0x3456,bw16> MY_PORT1;
uint16_t d;
//...
d = iord(MY_PORT1);   // no-overhead in-line code
iowr(MY_PORT1, 0x456);
```

If the symbolic port name is referenced via a pointer, then an *access_type* object must be instantiated in memory; (slower) generic functions are invoked by the iord(..)/iowr(..)/etc. functions.

In this case, the *access_type* parameter is mostly evaluated at runtime. (This approach is similar to the one used for *extern inline* functions in C)

Example:

```
typedef MM_DIRECT<uint16_t,rmw_t,chip8l,0x3456,bw16> MY_PORT1;
typedef MM_DIRECT<uint16_t,ro_t,chip16l,0x7890,bw16> MY_PORT2;

uint16_t foo(MM_DIRECT * iop)
    {
    return iord(iop); // invoke some generic iord function
    }

uint16_t a;
a = foo(MY_PORT1);
a +=foo(MY_PORT2);
```

C.5 Embedded systems extended memory support

Many embedded systems include memory that can only be accessed with some form of device driver. These include memories accessed by serial data busses (I2C, SPI), and on-board non-volatile memory. Device driver memory support is used in applications where the details of the access method can be separated from the details of the application.
In contrast to memory-mapped I/O, the extended memory layout and its use should be administrated by the compiler/linker.
Language support for embedded systems need to address the following issues:

1. Memory with user-defined device drivers. User-defined device drivers are required for reading and writing user-defined memory.
   - Memory read functions take as an argument an address in the user-defined memory space, and return data of a user-defined size.
   - Memory write takes two arguments an address in the user-defined memory space and data with a user-specified size.
   - Applications require support for multiple user-defined address spaces.
   - User-defined memory areas may not be contiguous. Most of the applications have gaps in the addressing within user-defined memory areas.

2. The compiler is responsible for:
   - Allocating variables, according to the needs of the application, in "normal" address space and in space accessed by the user-defined memory device drivers.
   - Making calls to device drivers, when accessing variables supported by user-defined device drivers.
   - Automating the process of casting and accessing the data, between calls to access data and the application.

3. Application variables in user-defined memory areas :
   - Need to support all of the available data types. For example, declarations for fundamental data types, arrays, structures.
   - Users need to direct the compiler to use a specific memory area.

- The compiler needs to be free to use user-defined memory area as a generic, general-purpose memory area, for the purposes of a variable spill area.

The following declaration shows all the information that is needed to declare memory for use with user-defined device drivers.

```
typemod USER_MEMORY <
   access limitation type ,
   device driver for data read,
   device driver for write,
   mary address constant,   // Base address of memory in
                            // the drivers address space
   address range            // Size of memory handled by
                            // the device drivers
  [optional additional address range definitions]
 > memory_name;
```

The *typemod* definition is a method of encapsulating the memory declaration. typemod ties variable declarations to device drivers, and provides the compiler a means of using data that the user provides to manage variables that are required by an application. User-defined memory may be global in nature, or local to one program segment.

```
typemod USER_MEMORY <rmw_t,
                     ddram_r,
                     ddram_w,
                     0x90,0x30
                     0xD0,0x30
                     > ddram

/* A typemod definition always specifies a linear memory (fragment(s))*/

/* function prototypes to read and write user-defined memory.
    It is the responsibility of the device driver to transfer
    the number of bytes requested. An optimizing compiler
    can pass structures or data, and the driver will
    optimize the transfer */

void ddram_w( int location, char *src, int size);
void ddram_r( int location, char *desc, int size);

char a;   /* normal memory declarations */
int b;
long c;

// Modifier puts variable in user-named address space

char ddram wa;
int ddram wb;
long ddram wc;
char ddram ar[10];
unsigned int ddram wc;

wa = 0x33; // ddram_w called (must be stored as an int)
a = wa;    // ddram_r called once, MSB turncated
b = wb;    // ddram_r called once
c = wc;    // ddram_r called more than once (implementation defined)
```

C.6 Comments on syntax notation

The syntax notation is inspired by the syntax used for template-based implementations of *iohw* in C++. However, this does not in any way imply that C++ features like templates and overloading should be added to the language. It is merely a new syntax for specifying the complex entity of an *access_type* (complex in the sense that it must consist of multiple entities).

The advantages with the proposed notation are: that it can be made reasonable consistent across processor and bus architectures, and (most importantly) it will be both fairly easy to comprehend and to use for the average embedded programmer. (In contrast to this are pure macro-based implementations, which tend to become rather complex to understand, create, and maintain for the user.)

The header file which defines the hardware will look simple (typically, like a list, with one register definition per text line). This makes it easy for a user to adapt an existing *access_type* definition to new hardware. Maintenance becomes much simpler.

Annex D

Migration path for iohw.h implementations.

(Informative annex)

D.1 Migration path for iohw.h implementations

It may take some time before compilers have full featured support for *access_types* based on intrinsic functionality. Until then efficient iohw implementations with a limited feature set can be implemented using C macros. This enable new I/O driver functions based on the iohw interface for basic I/O hardware addressing to be used with existing older compilers.

iohw.h implementation example based on C macros

The follow example illustrates how a simple but efficient iohw implementation can be created.

This implementation provide these recommended features:
- I/O access to 8,16 and 32 bit registers.
- Direct I/O access as memory mapped I/O
- I/O access via (intrinsic or user) access driver functions.
- I/O buffer access with register interleave.

This implementation does not provide these features:
- Support for bit access.
- Register multi-addressing and I/O register endian (register widths larger than the I/O chip bus width)
- More advance addressing methods (other that what can be implemented via access driver functions)
- Pass of access_type parameters between functions.

D.2.1 The iohw.h header

The first part implements the iohw macro functions and include stdint.h. The OR, AND, and XOR are here implemented as RD-modify-WR operations on the C source level.
Then a number of access_type macros are defined for memory mapped I/O access and access via (intrinsic) driver functions. The purpose of these macros is to simplify the users I/O hardware register definitions. New access methods can be added along the same line.

```
//********************** Start of IOHW ***************************
# ifndef IOHW_H
# define IOHW_H

// Define standard function macros for I/O hardware access
#define iord(  NAME )     ( NAME##_RDFUNC( NAME##_ADR ))
#define iowr(  NAME, VAL) ( NAME##_WRFUNC( NAME##_ADR, (VAL)))
#define ioor(  NAME, VAL) ( NAME##_WRFUNC( NAME##_ADR, (NAME##_RDFUNC(NAME##_ADR) | ( VAL ))))
#define ioand( NAME, VAL) ( NAME##_WRFUNC( NAME##_ADR, (NAME##_RDFUNC(NAME##_ADR) & ( VAL ))))
```

```
#define ioxor( NAME, VAL) ( NAME##_WRFUNC( NAME##_ADR, (NAME##_RDFUNC(NAME##_ADR) ^ ( VAL ))))

#define iordbuf(  NAME, INDEX )     ( NAME##_RDFUNC( NAME##_ADR+(INDEX)*NAME##_INTL))
#define iowrbuf(  NAME, INDEX, VAL) ( NAME##_WRFUNC( NAME##_ADR+(INDEX)*NAME##_INTL, (VAL)))
#define ioorbuf(  NAME, INDEX, VAL) ( NAME##_WRFUNC( NAME##_ADR+(INDEX)*NAME##_INTL,\
                                      (NAME##_RDFUNC( NAME##_ADR+(INDEX)*NAME##_INTL) | ( VAL ))))
#define ioandbuf( NAME, INDEX, VAL) ( NAME##_WRFUNC( NAME##_ADR+(INDEX)*NAME##_INTL, \
                                      (NAME##_RDFUNC( NAME##_ADR+(INDEX)*NAME##_INTL) & ( VAL ))))
#define ioxorbuf( NAME, INDEX, VAL) ( NAME##_WRFUNC( NAME##_ADR+(INDEX)*NAME##_INTL, \
                                      (NAME##_RDFUNC( NAME##_ADR+(INDEX)*NAME##_INTL) ^ ( VAL ))))

#define io_at_init( NAME )       NAME##_INIT
#define io_at_release( NAME )    NAME##_RELEASE
// the only dynamic access_type parameter in this implementation is the address
#define io_at_cpy( DNAME, SNAME ) ((DNAME##_ADR) = (SNAME##_ADR))

// Include standard integer type definitions similar to:
// #define uint8_t  unsigned char
// #define uint16_t unsigned short
// #define uint32_t unsigned long
#include <stdint.h>

// Some access_type macros which simplify the users access_types definitions
// These macros are platform specfic (intrinsic)
//  There should be one definition for each addressing range, register width and access method
supported

// Some access macros for simple memory mapped I/O
#define RDPTR8(address)       (* (volatile uint8_t  *) ( address ))
#define WRPTR8(address,val)  ((* (volatile uint8_t  *) ( address )) = (uint8_t)(val))
#define RDPTR16(address)      (* (volatile uint16_t *) ( address ))
#define WRPTR16(address,val) ((* (volatile uint16_t *) ( address )) = (uint16_t)(val))
#define RDPTR32(address)      (* (volatile uint32_t *) ( address ))
#define WRPTR32(address,val) ((* (volatile uint32_t *) ( address )) = (uint32_t)(val))

// Some access macros for access via (intrinsic) I/O hardware access functions
#define RDFUNC8(address)      (_inp(  (uint16_t)(address)))
#define WRFUNC8(address,val)  (_outp( (uint16_t)(address),(uint8_t)(val)))
#define RDFUNC16(address)     (_inpw( (uint16_t)(address)))
#define WRFUNC16(address,val) (_outpw((uint16_t)(address),(uint16_t)(val)))
#define RDFUNC32(address)     (_inpd( (uint16_t)(address)))
#define WRFUNC32(address,val) (_outpd((uint16_t)(address),(uint32_t)(val)))

// Prototype for the (intrinsic) I/O hardware access functions
uint8_t  _inp(  uint16_t adr);
uint16_t _inpw( uint16_t adr);
uint32_t _inpd( uint16_t adr);
void _outp( uint16_t adr, uint8_t val);
void _outpw(uint16_t adr, uint16_t val);
void _outpd(uint16_t adr, uint32_t val);

# endif
//********************** End of IOHW ***************************
```

D.2.2 The users I/O register definitions

For each I/O register (each symbolic name) a complete definition of the access method must be created. With this iohw implementation the user must define:

- The access method for RD and/or WR operations (which also define the register size and select between memory mapped I/O or access via driver functions).
- The I/O register address
- The interleave factor for register arrays.

This platform dependent I/O register definitions are normally placed in a separate header file. Here called iohw_ta.h.

```
//****** Start of user I/O register definitions (IOHW_TA.H) ******
#ifndef IOHW_TA
#define IOHW_TA

#define  MYPORTS_INIT    {/* No initialization needed in this system */}
#define  MYPORTS_RELEASE {/* No release needed in this system */}

// REG_A register is accessed via a base address
extern uint16_t base_adr;    // is placed in some platform dependent C module
#define  REG_A_ADR     base_adr
#define  REG_A_WRFUNC  WRFUNC8
#define  REG_A_RDFUNC  RDFUNC8

// MYPORT1 is a bidirectional memory mapped register
#define  MYPORT1_ADR    0x1234    // Defines physical address
#define  MYPORT1_WRFUNC  WRPTR8   // Defines access method, bus type and register size
#define  MYPORT1_RDFUNC  RDPTR8

// MYPORT2 is a write-only memory mapped register
// iord(),ioor(),ioand(),ioxor() will produce a compiler warning
#define  MYPORT2_ADR    0x4567
#define  MYPORT2_WRFUNC  WRPTR16  // Only write operations is defined

// MYPORT3 is a read-only memory mapped register
// iowr(),ioor(),ioand(),ioxor() will produce a compiler warning
#define  MYPORT3_ADR    0x7870
#define  MYPORT3_RDFUNC  RDPTR16  // Only read operations is defined

// MYPORT4 is a bidirectional port accessed via (intrinsic) driver functions
#define  MYPORT4_ADR    0x12
#define  MYPORT4_WRFUNC  WRFUNC8
#define  MYPORT4_RDFUNC  RDFUNC8

// MYPORT5 is a bidirectional memory mapped register array
#define  MYPORT5_ADR    0x1234    // Defines physical address
#define  MYPORT5_WRFUNC  WRPTR8   // Defines access method, bus type and register size
#define  MYPORT5_RDFUNC  RDPTR8
#define  MYPORT5_INTL    2        // 16 bit processor bus / 8 bit I/O chip bus

// MYPORT6 is yet another register of the same type as MYPORT4
#define  MYPORT6_ADR     0x234
#define  MYPORT6_WRFUNC  WRFUNC8
#define  MYPORT6_RDFUNC  RDFUNC8

#endif
//************ End of user I/O register defintions ********************
```

D.2.3 The driver function

The driver function should include iohw.h and the user I/O register definitions for the target system iohw_ta.h. The example below tests all operation on all the previous I/O register definitions.
Note that illegal I/O register access operations in the driver code will be detected at compile time ex. read of a write-only register or write of a read-only register.

```
#include <iohw.h>    // includes stdint.h
#include <iohw_ta.h> // My register definitions

uint8_t cdat;
uint16_t idat;

uint16_t my_func(void)
   {
   uint16_t t = (uint16_t) iord(REG_A);
   iowr(REG_A,0x80);
   return t;
   }

void my_test_driver (void)
   {
   io_at_init(MYPORTS);

   // Test bidirectional memory mapped I/O port
   cdat = iord(MYPORT1);
   iowr(MYPORT1,0x12);
   iowr(MYPORT1,cdat);
   ioor(MYPORT1, 0x23);
   ioand(MYPORT1,0x34);
   ioxor(MYPORT1,0xf0);

   // Test unidirectional I/O ports
   iowr(MYPORT2,0x3455);
   idat = iord(MYPORT3);

   // Test bidirectional I/O register accessed via functions
   cdat = iord(MYPORT4);
   iowr(MYPORT4,0x12);
   iowr(MYPORT4,cdat);
   ioor(MYPORT4, 0x23);
   ioand(MYPORT4,0x34);
   ioxor(MYPORT4,0xf0);

   // Test buffer functions
   cdat = iordbuf(MYPORT5,20);
   iowrbuf(MYPORT5,43,0x12);
   ioorbuf(MYPORT5,43,0x12);
   ioandbuf(MYPORT5,43,0x02);
   ioxorbuf(MYPORT5,43,0x12);

   // These statements will produce compiler warnings
   // idat = iord(MYPORT2); // Error write-only port
   // iowr(MYPORT3,4565);   // Error read-only port

   // REG_A, MYPORT4 and MYPORT6 must be of the same type
   io_at_cpy(REG_A, MYPORT4);
   idat = my_func();        // Access MYPORT4 via driver
   io_at_cpy(REG_A, MYPORT6);
   idat = my_func();        // Access MYPORT6 via driver
   io_at_release(MYPORTS);
   }
```