

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

**REVIEW VERSION FOR DD 2003-02-26 FOR PDTR APPROVAL BALLOT**

**Contents**

1	GENERAL .....	6
1.1	Scope.....	6
1.2	References .....	6
2	FIXED-POINT ARITHMETIC.....	7
2.1	Overview and principles of the fixed-point data types.....	7
2.1.1	The data types.....	7
2.1.2	Spelling of the new keywords.....	8
2.1.3	Overflow and Rounding.....	9
2.1.4	Type conversion, usual arithmetic conversions .....	10
2.1.5	Fixed-point constants .....	11
2.1.6	Operations involving fixed-point types.....	11
2.1.7	Fixed-point functions .....	14
2.1.8	Fixed-point definitions <stdfix.h>.....	16
2.1.9	Formatted I/O functions for fixed-point arguments.....	16
2.2	Detailed changes to ISO/IEC 9899:1999 .....	16
3	NAMED ADDRESS SPACES AND NAMED-REGISTER STORAGE CLASSES...40	40
3.1	Overview and principles of named address spaces .....	40
3.1.1	Additional address spaces .....	40
3.1.2	Address-space type qualifiers .....	40
3.1.3	Address space nesting and rules for pointers .....	41
3.1.4	Standard library support .....	42

# ISO/IEC PDTR 18037

3.2 Overview and principles of named-register storage classes .....	42
3.2.1 Access to machine registers .....	42
3.2.2 Named-register storage-class specifiers .....	42
3.2.3 Ensuring correct side effects via objects allocated in registers .....	44
3.2.4 Relationship between named registers and I/O-register designators .....	44
3.3 Impact on the C language usage. ....	44
<b>4 BASIC I/O HARDWARE ADDRESSING.....</b>	<b>52</b>
4.1 Rationale.....	52
4.1.1 Basic Standardisation Objectives.....	52
4.2 Concepts.....	52
4.3 Basic I/O-Hardware addressing header <iohw.h> .....	54
4.3.1 Standardisation principles .....	54
4.3.2 The abstract model.....	54
4.4 Specifying I/O registers .....	56
4.4.1 I/O-register designators .....	56
4.4.2 Accesses to individual I/O registers .....	57
4.4.3 I/O register buffers.....	58
4.4.4 I/O groups.....	59
4.4.5 Direct and indirect designators.....	59
4.4.6 Operations on I/O groups .....	60
4.5 Detailed changes to ISO/IEC 9899:1999 .....	61
<b>ANNEX A.....</b>	<b>68</b>
A.1 Fixed-point .....	68
A.1.1 The fixed-point data types .....	68
A.1.2 Classification of Fixed Point Types .....	69
A.1.3 Recommendations for Fixed Point Types in C .....	70
A.1.4 Number of Fractional data bits in _Fract versus _Accum .....	71
A.1.5 Possible Data Type Implementations.....	71
A.1.6 Overflow and Rounding .....	72
A.1.7 Type conversions, usual arithmetic conversions.....	73
A.1.8 Operations involving fixed-point types.....	74
A.1.9 Exception for 1 and -1 Multiplication Results .....	75
A.1.10 Linguistic Variables and unsigned _Fract: an example of unsigned fixed-point.....	76
<b>ANNEX B.....</b>	<b>77</b>
B.1 Embedded systems extended memory support .....	77
B.1.1 Modifiers for named address spaces .....	77
B.1.2 Application-defined multiple address space support.....	78

ANNEX C .....	80
C.1General .....	80
C.1.1 Recommended steps.....	80
C.1.2 Compiler considerations.....	80
C.2Overview of I/O Hardware Connection Options .....	81
C.2.1 Multi-Addressing and I/O Register Endian.....	81
C.2.2 Address Interleave.....	82
C.2.3 I/O Connection Overview: .....	82
C.2.4 Generic buffer index .....	83
C.3I/O-register designators for different I/O addressing methods .....	84
C.4Atomic operation.....	85
C.5Read-modify-write operations and multi-addressing cases. ....	85
C.6I/O initialisation .....	86
C.7Intrinsic Features for I/O Hardware Access.....	87
ANNEX D .....	88
D.1Generic access_spec descriptor .....	88
D.1.1 Background .....	88
D.2Syntax specification.....	88
D.3Examples of access_spec descriptors .....	90
D.4Parsing.....	92
D.5Comments on syntax notation .....	93
ANNEX E.....	94
E.1Migration path for <iohw.h> implementations.....	94
E.2<iohw.h> implementation based on C macros .....	94
E.2.1 The access specification method .....	94
E.2.2 An <iohw.h> implementation technique.....	95
E.2.3 Features .....	95
E.2.4 The <iohw.h> header.....	96
E.2.5 The users I/O-register designator definitions .....	98
E.2.6 The driver function.....	100
ANNEX F.....	101

# **ISO/IEC PDTR 18037**

F.1 Circular buffers .....	101
F.2 Complex data types.....	102
F.3 Consideration of BCD data types for Embedded Systems .....	102
F.4 Modwrap overflow .....	103
 ANNEX G .....	104
G.1 Compatibility with C++.....	104
G.1.1 Fixed-point Arithmetic.....	104
G.1.2 Multiple Address Spaces Support .....	104
G.1.3 Basic I/O Hardware Addressing .....	104

## 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 data type used in filtering algorithms is the fixed-point data type, and in order to achieve the necessary speed, the embedded processors are often equipped with special hardware support that data type. The C language (as defined in ISO/IEC 9899:1999) does not provide support the fixed-point arithmetic operations, currently leaving programmers with no option but to handcraft most of their algorithms in assembler. This Technical Report specifies a fixed-point data type 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 basic support for multiple address spaces. As a result, optimising compilers can utilise the ability of processors that support multiple address spaces, for instance, to read data from two separate memories in a single cycle to maximise execution speed.

As the C language has matured over the years, various extensions for accessing basic I/O hardware (*iohw*) registers have been added to address deficiencies in the language. Today almost all C compilers for freestanding environments and embedded systems support some method of direct access to *iohw* registers from the C source level. However, these extensions have not been consistent across dialects.

This Technical Report provides an approach to codifying common practice and providing a single uniform syntax for basic *iohw* 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 data types

#### 2.1.1 The data types

For the purpose of this Technical Report, fixed-point data values are either fractional data values (with value between -1.0 and +1.0), or data values with an integral part and a fractional part. As the position of the radix point is known implicitly, operations on the values of these data types can be implemented with (almost) the same efficiency as operations on integral values. Typical usage of fixed-point data values 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 data types, see clause A.1.1 in the Annex of this Technical Report.

For the purpose of this Technical Report, two groups of fixed-point data types are added to the C language: the *fract types* and the *accum types*. The data value of a fract type has no integral part, hence values of a fract type are between -1.0 and +1.0. The value range of an accum type depends on the number of integral bits in the data type.

The fixed-point data types are designated with the corresponding new keywords and *type-specifiers* **\_Fract** and **\_Accum**. These *type-specifiers* can be used in combination with the existing *type-specifiers* **short**, **long**, **signed** and **unsigned** to designate the following twelve fixed-point types:

<b>unsigned short _Fract</b>	<b>unsigned short _Accum</b>
<b>unsigned _Fract</b>	<b>unsigned _Accum</b>
<b>unsigned long _Fract</b>	<b>unsigned long _Accum</b>
<b>signed short _Fract</b>	<b>signed short _Accum</b>
<b>signed _Fract</b>	<b>signed _Accum</b>
<b>signed long _Fract</b>	<b>signed long _Accum</b>

The fixed-point data types

<b>short _Fract</b>	<b>short _Accum</b>
<b>_Fract</b>	<b>_Accum</b>
<b>long _Fract</b>	<b>long _Accum</b>

without either **unsigned** or **signed** are aliases for the corresponding signed fixed-point types.

An implementation is required to support all above mentioned twelve fixed-point data types. Just as for integer types, there is no requirement that the types all have different formats.

The fixed-point types are assigned a *fixed-point rank*. The following types are listed in order of increasing rank:

`short _Fract, _Fract, long _Fract, short _Accum, _Accum, long _Accum`

Each unsigned fixed-point type has the same size (in bytes) and the same rank as its corresponding signed fixed-point type.

The bits of an unsigned fixed-point type are divided into padding bits, fractional bits, and integral bits. The bits of a signed fixed-point type are divided into padding bits, fractional bits, integral bits, and a sign bit.

The fract fixed-point types have no integral bits; consequently, values of unsigned fract types are in the range of 0 to 1, and values of signed fract types are in the range of -1 to 1. The minimal formats for each type are:

<code>signed short _Fract</code>	<code>s.7</code>	<code>signed short _Accum</code>	<code>s4.7</code>
<code>signed _Fract</code>	<code>s.15</code>	<code>signed _Accum</code>	<code>s4.15</code>
<code>signed long _Fract</code>	<code>s.23</code>	<code>signed long _Accum</code>	<code>s4.23</code>
<code>unsigned short _Fract</code>	<code>.7</code>	<code>unsigned short _Accum</code>	<code>4.7</code>
<code>unsigned _Fract</code>	<code>.15</code>	<code>unsigned _Accum</code>	<code>4.15</code>
<code>unsigned long _Fract</code>	<code>.23</code>	<code>unsigned long _Accum</code>	<code>4.23</code>

(For the unsigned formats, the notation "x.y" means x integral bits and y fractional bits, for a total of x + y non-padding bits. The added "s" in the signed formats denotes the sign bit.)

An implementation may give any of the fixed-point types more fractional bits, and may also give any of the accum types more integral bits; the relevant restrictions are given in the text for the new section 6.2.5 (see section 2.2 of this Technical Report).

For an example of unsigned fixed-point datatypes see A.1.10.

### 2.1.2 Spelling of the new keywords

The natural spelling of the newly introduced keywords `_Fract` and `_Accum`, and of the new type-qualifier `_Sat`, is `fract`, `accum` and `sat`. However, in order to avoid nameclashes in existing programs the new keywords are handled in the same way as the `_Complex` keyword in the ISO/IEC 9899:1999 standard: the formal names of the new keywords start with an underscore, followed by a capital letter, and in the for fixed-point arithmetic required header `<stdfix.h>`, these formal names can be redefined to have their natural spelling, or to another spelling, for instance, in an environment with pre-existing fixed-point support.

In the code fragments in this Technical Report, the natural spelling will be used.

For information on the usage of the new keywords in a combined C/C++ environment, see Annex G.

### 2.1.3 Overflow and Rounding

Conversion of a real numeric value to a fixed-point type may overflow and/or may require rounding. When the source value does not fit within the range of the fixed-point type, the conversion overflows. Of the two common approaches for fixed-point overflow handling (saturation and modular wrap-around) only saturation is required by this Technical Report; for a description of modular wrap-around, see section F.4.

When calculating the saturated result on fixed-point overflow, the source value is replaced by the closest available fixed-point value. (For unsigned fixed-point types, this will be either zero or the maximal positive value of the fixed-point type. For signed fixed-point types it will be the maximal negative or maximal positive value of the fixed-point type.)

Overflow behavior is controlled in two ways:

- A *fixed-point overflow type-qualifier* **\_Sat** can be added to a fixed-point type to control overflow behavior (e.g., **\_Sat \_Fract**).
- In the absence of an explicit fixed-point overflow type-qualifier, overflow behavior is controlled by the **FX\_FRACT\_OVERFLOW** and **FX\_ACCUM\_OVERFLOW** pragmas with **SAT** and **DEFAULT** as possible states.

When the state of the **FX\_FRACT\_OVERFLOW** pragma is **SAT**, the overflow behaviour on **\_Fract** types is saturation; otherwise, overflow on **\_Fract** types has undefined behavior.

When the state of the **FX\_ACCUM\_OVERFLOW** pragma is **SAT**, the overflow behaviour on **\_Accum** types is saturation; otherwise, overflow on **\_Accum** types has undefined behavior.

The default state for the **FX\_FRACT\_OVERFLOW** and **FX\_ACCUM\_OVERFLOW** pragmas is **DEFAULT**.

Note: the **DEFAULT** state of the overflow pragmas is intended to allow implementations to use the most optimal instruction sequences irrespective of their overflow behaviour for those computations where the actual overflow behaviour is not relevant; the actual overflow behaviour may be saturation, or anything else (including modular wrap-around) and may vary between different occurrences of the same operation, or even between different executions of the same operation.

If (after any overflow handling) the source value cannot be represented exactly by the fixed-point type, the source value is rounded to either the closest fixed-point value greater than the source value (rounded up) or to the closest fixed-point value less than the source value (rounded down).

Processors that support fixed-point arithmetic in hardware have no problems in attaining the required precision without loss of speed; however, simulations using integer arithmetic may require for multiplication and division extra instructions to get correct result; often these additional instructions are not needed if the required precision is 2 ulps<sup>1</sup>. The **FX\_FULL\_PRECISION** pragma provides a means to inform the implementation when a program requires full precision for these operations (the state of the **FX\_FULL\_PRECISION** pragma is "on"), or when the relaxed

---

<sup>1</sup> unit in the last place: precision up to the last bit

requirements are allowed (the state of the **FX\_FULL\_PRECISION** pragma is "off"). For more discussion on this topic see A.1.6.

Whether rounding is up or down is implementation-defined and may differ for different values and different situations.

#### 2.1.4 Type conversion, usual arithmetic conversions

All conversions between a fixed-point type and another arithmetic type (which can be another fixed-point type) are defined. Overflow and rounding are handled according to the usual rules for the destination type. Conversions from a fixed-point to an integer type round toward zero. The rounding of conversions from a fixed-point type to a floating-point type is unspecified.

The usual arithmetic conversions in the C standard (see 6.3.1.8) imply three requirements:

1. given a pair of data types, the usual arithmetic conversions define the *common type* to be used;
2. then, if necessary, the usual arithmetic conversions require that each operand is converted to that common type; and
3. it is required that the resulting type after the operation is again of the common type.

For the combination of an integer type and a fixed-point type, or the combination of a fract type and an accum type the usual arithmetic rules may lead to useless results (converting an integer to a fixed-point type) or to gratuitous loss of precision.

In order to get useful and attainable results, the usual arithmetic conversions do not apply to the combination of an integer type and a fixed-point type, or the combination of two fixed-point types.

In these cases:

1. the result of the operation is calculated using the values of the two operands, with their full precision; and
2. the result type is the type with the highest rank, whereby a fixed-point conversion rank is always greater than an integer conversion rank, and the resulting value is converted (taking into account rounding and overflow) to the precision of the resulting type;
3. if one operand has signed fixed-point type and the other operand has unsigned fixed-point type, then the resulting type is the signed type corresponding to the operand type with greatest fixed-point conversion rank.

Note that as a consequence of the above, in the following fragment

```
fract r, r1, r2; int i;  
  
r1 = r * i; r2 = r * (fract) i;
```

the result values **r1** and **r2** may not be the same.

If the type of either of the operands has the **\_Sat** qualifier, the resulting type shall have the **\_Sat** qualifier.

If one of the operands has a floating type and the other operand has a fixed-point type, the result of the operation is calculated using the values of the two operands, with their full precision and the result type is the floating type.

It is recommended that a conforming compilation system provide an option to produce a diagnostic message whenever the usual arithmetic conversions cause a fixed-point operand to be converted to floating-point.

## 2.1.5 Fixed-point constants

A *fixed-constant* is defined analogous to a floating-constant (see 6.4.4.2), with suffixes **k** (**K**) and **r** (**R**) for accum type constants and fract type constants; for the short variants the suffix **h** (**H**) should be added as well.

The type of a fixed-point constant depends on its *fixed-suffix* as follows (note that the suffix is case insensitive; the table below only give lowercase letters):

Suffix	Fixed-point type
<b>hr</b>	<b>short _Fract</b>
<b>uh</b> <b>r</b>	<b>unsigned short _Fract</b>
<b>r</b>	<b>_Fract</b>
<b>ur</b>	<b>unsigned _Fract</b>
<b>lr</b>	<b>long _Fract</b>
<b>ul</b> <b>r</b>	<b>unsigned long _Fract</b>
<b>hk</b>	<b>short _Accum</b>
<b>uhk</b>	<b>unsigned short _Accum</b>
<b>k</b>	<b>_Accum</b>
<b>uk</b>	<b>unsigned _Accum</b>
<b>lk</b>	<b>long _Accum</b>
<b>ulk</b>	<b>unsigned long _Accum</b>

A fixed-point constant shall evaluate to a value that is in the range for the indicated type. An exception to this requirement is made for constants of one of the fract types with value **1**; these constants shall denote the maximal value for the type.

## 2.1.6 Operations involving fixed-point types

### 2.1.6.1 Unary operators

#### 2.1.6.1.1 Prefix and postfix increment and decrement operators

The prefix and postfix `++` and `--` operators have their usual meaning of adding or subtracting the integer value 1 to or from the operand and returning the value before or after the addition or subtraction as the result.

## 2.1.6.1.2 Unary arithmetic operators

The unary arithmetic operators plus (`+`) and negation (`-`) are defined for fixed-point operands, with the result type being the same as that of the operand. The negation operation is equivalent to subtracting the operand from the integer value zero. It is not allowed to apply the complement operator (`~`) to a fixed-point operand. The result of the logical negation operator `!` applied to a fixed-point operand is 0 if the operand compares unequal to 0, 1 if the value of the operand compares equal to 0; the result has type `int`.

## 2.1.6.2 Binary operators

### 2.1.6.2.1 Binary arithmetic operators

The binary arithmetic operators `+`, `-`, `*`, and `/` are supported for fixed-point data types, with their usual arithmetic meaning, whereby the usual arithmetic conversions for expressions involving fixed-point type operands, as described in section 2.1.4, are applied.

If the result type of an arithmetic operation is a fixed-point type, for operators other than `*` and `/`, the calculated result is the mathematically exact result with overflow handling and rounding performed to the full precision of the result type as explained in the earlier section on Overflow and Rounding. The `*` and `/` operators may return either this rounded result or, depending of the state of the `FX_FULL_PRECISION` pragma, the closest larger or closest smaller value representable by the result fixed-point type. (Between rounding and this optional adjustment, the multiplication and division operations permit a mathematical error of almost 2 units in the last place of the result type.)

If the mathematical result of the `*` operator is exactly `1`, the closest smaller value representable by the fixed point result type may be returned as the result, even if the result type can represent the value `1` exactly. Correspondingly, if the mathematical result of the `*` operator is exactly `-1`, the closest larger value representable by the fixed point result type may be returned as the result, even if the result type can represent the value `-1` exactly. The circumstances in which a `1` or `-1` result might be replaced in this manner are implementation-defined.

Note that the special treatment of the values `1` and `-1` as result of a `*` operation, as indicated in this paragraph, is only included to ensure that certain specific implementations, that otherwise would not be conformant, can conform to this Technical Report; it is the intention to deprecate this treatment in future revisions of this specification. For more discussion, see Annex A.1.9.

If the value of the second operand of the `/` operator is zero, the behaviour is undefined.

According to the rules above, the result type of an arithmetic operation where (at least) one operand has a fixed-point type is always a fixed-point type. Other combinations of operand and result types for arithmetic operations are supported through special functions. The generic function names for these functions are `mulifx`, `divifx`, `fxdivi` and `idivifx`, where `fx` stands for one of `r`, `lr`,

**k, lk, ur, ulr, uk and ulk.** The **muli** functions perform a multiply of an integer operand and a fixed-point operand and return an integer value; the **divi** functions divide the first (integer) operand by a fixed-point operand (**divifx**) yielding a fixed-point type result, or divide two integer operands (**fxdivi**) yielding a fixed-point type result; the **idivfx** functions divide the two fixed-point type operands (with the same type) and return an integer result.

#### 2.1.6.2.2 Bitwise shift operators

Shifts of fixed-point values using the standard << and >> operators are defined to be equivalent to multiplication or division by a power of two (including the resulting overflow and rounding behavior). The right operand is converted to type **int** and must be nonnegative and less than the total number of (nonpadding) bits of the fixed-point operand (the left operand). The result type is the same as that of the fixed-point operand. An exact result is calculated and then converted to the result type in the same way as the other fixed-point arithmetic operators.

#### 2.1.6.2.3 Relational operators, equality operators

The standard relational operators (<, <=, >=, and >) and equality operators (==, and !=) accept fixed-point operands. When comparing fixed-point values with fixed-point values or integer values, the values are compared directly; the values of the operands are not converted before the comparison is made. Otherwise, the usual arithmetic conversions are applied before the comparison is made.

#### 2.1.6.3 Assignment operators

The standard assignment operators +=, -=, \*=, and /= are defined in the usual way when either operand is fixed-point.

The standard assignment operators <<= and >>= are defined in the usual way when the left operand is fixed-point.

#### 2.1.6.4 Example of fixed-point usage

The following example calculating a scaled dot-product of two **fract** vectors clarifies how the fixed-point features are intended to be used.

Example:

```
fract a[N], b[N], z;
long accum acc = 0;
for ( int ix = 0; ix < N; ix++ ) {
    acc += (long accum) a[ix] * b[ix];
}
z = acc >> SCALE;
```

This example is without any explicit rounding and with default overflow handling.

## 2.1.7 Fixed-point functions

### 2.1.7.1 The fixed-point absolute value functions

The absolute value functions **absfx**, where *fx* stands for one of **hr**, **r**, **lr**, **hk**, **k** or **lk**, take one fixed-point type argument (corresponding to *fx*); the function type is the same as the type of the argument.

The absolute value functions compute the absolute value of a fixed-point value. If the exact result value cannot be represented, the saturated result value is returned.

### 2.1.7.2 The fixed-point rounding functions

The rounding functions **roundfx**, where *fx* stands for one of **hr**, **r**, **lr**, **hk**, **k** or **lk**, take two arguments: a fixed-point argument (corresponding to *fx*) and an integer argument; the function type is the same as the type of the first argument. Similarly the rounding functions **roundufx** take an unsigned fixed-point argument as first argument, and the function type is the same unsigned fixed-point type.

The value of the second argument must be nonnegative and less than the number of fractional bits in the fixed-point type of the first argument. The rounding functions compute the value of the first argument, rounded to the number of fractional bits specified in the second argument. The rounding applied is to-nearest, with unspecified rounding direction in the halfway case. Fractional bits beyond the rounding point are set to zero in the result. If the exact result value cannot be represented, the saturated result value is returned.

### 2.1.7.3 The fixed-point bit countls functions

The bit count functions **countlsfx**, where *fx* stands for one of **hr**, **r**, **lr**, **hk**, **k**, **lk**, **uhr**, **ur**, **ulr**, **uhk**, **uk** or **ulk**, take one fixed-point type argument (corresponding to *fx*); the function type is **int**.

The integer return value of the above functions is defined as follows:

- if the value of the fixed-point argument is non-zero, the return value is the largest integer **k** for which the expression **a<<k** does not overflow;
- if the value of the fixed-point argument is zero, an integer value is returned that is at least as large as N-1, where N is the total number of (nonpadding) bits of the fixed-point type of the argument.

### 2.1.7.4 The bitwise fixed-point to integer conversion functions

The bitwise fixed-point to integer conversion functions **bitsfx**, where *fx* stands for one of **hr**, **r**, **lr**, **hk**, **k**, **lk**, **uhr**, **ur**, **ulr**, **uhk**, **uk** or **ulk**, take one fixed-point type argument (corresponding to *fx*); the type of the function is an implementation-defined integer type **int\_fx\_t**, defined in the **<stdfix.h>** headerfile, that is large enough to hold all the bits in the fixed-point type.

The bitwise fixed-point to integer conversion functions return an integer value equal to the fixed-point value of the argument multiplied by  $2^F$ , where F is the number of fractional bits in the fixed-point type. The result type is an integer type big enough to hold all valid result values for the given fixed-point argument type. For example, if the **fract** type has 15 fractional bits, then after the declaration

```
fract a = 0.5;
```

the value of **bitsr(a)** is  $0.5 * 2^{15} = 0x4000$ .

#### 2.1.7.5 The bitwise integer to fixed-point conversion functions

The bitwise fixed-point to integer conversion functions **fxbits**, where **fx** stands for one of **hr**, **r**, **lr**, **hk**, **k**, **lk**, **uhr**, **ur**, **ulr**, **uhk**, **uk** or **ulk**, take one argument with type **int** or **unsigned int**, the function type is a fixed-point type (corresponding to **fx**).

The bitwise fixed-point to integer conversion functions return an fixed-point value equal to the integer value of the argument divided by  $2^F$ , where F is the number of fractional bits in the fixed-point result type of the function. For example, if **fract** has 15 fractional bits, then the value of **rbits(0x2000)** is 0.25.

#### 2.1.7.6 Type-generic fixed-point functions

The header **<stdfix.h>** defines the following fixed-point type-generic macros. For each of the fixed-point absolute value functions in 2.1.7.1, the fixed-point round functions in 2.1.7.2, the fixed-point countls functions in 2.1.7.3, the functions for bitwise conversion of fixed-point to integer values in 2.1.7.4 and the functions for bitwise conversion of integer to fixed-point values in 2.1.7.5, a type-generic macro is defined as follows:

	type-generic macro
the fixed-point absolute value functions	<b>absfx</b>
the fixed-point round functions	<b>roundfx</b>
the fixed-point countls functions	<b>countlfx</b>

#### 2.1.7.7 Fixed-point numeric conversion functions

The fixed-point numeric conversion functions **strtofx**, where **fx** stands for one of **hr**, **r**, **lr**, **hk**, **k**, **lk**, **uhr**, **ur**, **ulr**, **uhk**, **uk** or **ulk**, take two arguments: the first argument has **const char \* restrict** type, the second argument has **char \*\* restrict** type; the function type is a fixed-point type (corresponding to **fx**).

Similar to the **strtod** function, the **strtofx** functions convert a portion of the string pointed to by the first argument to a fixed-point representation, with a type corresponding to **fx**, and return that fixed-point type value.

## 2.1.8 Fixed-point definitions <stdfix.h>

In the header <stdfix.h> defines macros that specify the precision of the fixed-point types and declares functions that support fixed-point arithmetic.

## 2.1.9 Formatted I/O functions for fixed-point arguments

Additional conversion specifiers for fixed-point arguments are defined as follows:

- r** for (signed) fract types
- R** for unsigned fract types
- k** for (signed) accum types
- K** for unsigned accum types.

Together with the standard length modifiers **h** (for short fixed-point arguments) and **l** (for long fixed-point arguments) all fixed-point types can be converted in the normal manner. Conversions to and from infinity and NaN representations are not supported.

The **fprintf** function and its derived functions with the **r**, **R**, **k** and **K** conversion formats convert the argument with a fixed-point type representing a fixed-point number to decimal notation in the style **[-]ddd.ddd**, where the number of digits after the decimal point is equal to the precision specification (i.e., it corresponds to the output format of the **f** (**F**) conversion specifier).

The **fscanf** function and its derived functions match an optionally signed fixed-point number whose format is the same as expected for the subject sequence of the corresponding **strtod** function. The corresponding argument of fscanf shall be a pointer to a fixed-point type variable with a type corresponding to **fx**.

## 2.2 Detailed changes to ISO/IEC 9899:1999

This section details the modifications to ISO/IEC 9899:1999 needed to incorporate the fixed-point functionality as described in section 2.1 of this Technical Report. The changes listed in this section are limited to syntax and semantics; examples, (forward) references and other descriptive information are omitted. The modifications are ordered according to the sections of ISO/IEC 9899:1999 to which they refer. If a section of ISO/IEC 9899:1999 is not mentioned, no changes to that section are needed. New sections are indicated with **(NEW SECTION)**, however resulting changes in the existing numbering are not indicated.

### Section 4 - Conformance

When incorporating the functionality, as described in section 2.1 of this Technical Report, the issue of conformance should be investigated: is it necessary to treat the fixed-point types the same as the complex types for a conforming freestanding implementation? Or are there other approaches to (partial) conformance?

### Section 5.2.4.2.3 - Characteristics of fixed-point types (NEW SECTION)

The characteristics of fixed-point data types are defined in terms of a model that describes a representation of fixed-point numbers and values that provide information about an implementation's fixed-point arithmetic. (The fixed-point model is intended to clarify the description of each fixed-point characteristic and does not require the fixed-point arithmetic of the implementation to be identical.)

Analogous to the Scaled data type, as defined in ISO/IEC 11404:1996 - Language-Independent Datatypes (LID), a *fixed-point number* ( $x$ ) is defined by the following model:

$$x = s * n * (b^f)$$

with the following parameters:

$s$	sign ( $\pm 1$ )
$b$	base or radix of nominator representation (an integer $> 1$ )
$p$	precision (the number of base- $b$ digits in the nominator)
$n$	nominator (nonnegative integer less than $b$ raised to the power $p$ )
$f$	factor (an integer value).

For the purpose of this Technical Report, the following restrictions to the above general model apply:

- $b$  equals 2: only binary fixed-point is considered;
- $(-p) \leq f < 0$ : integer values ( $f \geq 0$ ) are not considered to form part of the fixed-point values, and the radix *dot* is assumed to be between or immediately to the left of the most significant digit in the nominator.

Fixed-point infinities or NaNs are not supported.

For *fract fixed-point* types,  $f$  equals  $(-p)$ : values with (signed) fract fixed-point types are between -1 and 1, values with unsigned fract fixed-point types are between 0 and 1.

For *accum fixed-point* types,  $f$  is between  $(-p)$  and zero: the value range of accum fixed-point types depends on the number of integral bits ( $f - p$ ) in the type.

If the result type of an arithmetic operation is a fixed-point type, the operation is performed exact according to its mathematical definition, and then overflow handling and rounding is performed for the result type.

The fixed-point overflow behaviour is either saturation (see clause 6.7.3) or undefined.

Overflow behavior is controlled in two ways:

- A *fixed-point overflow type-qualifier* **\_Sat** can be added to a fixed-point type to control overflow behavior (e.g., **\_Sat \_Fract**).
- In the absence of an explicit fixed-point overflow type-qualifier, overflow behavior is controlled by the **FX\_FRACT\_OVERFLOW** pragma for **\_Fract** types and the **FX\_ACCUM\_OVERFLOW**

pragma for `_Accum` types. These pragmas follows the same scoping rules as existing `STDC` pragmas (see clause 6.10.6 of the C standard), and have the following syntax:

```
#pragma STDC FX_FRACT_OVERFLOW overflow-switch
#pragma STDC FX_ACCUM_OVERFLOW overflow-switch
```

where `overflow-switch` is one of `SAT` or `DEFAULT`.

When the state of the `FX_FRACT_OVERFLOW` pragma is `SAT`, the overflow behaviour on `_Fract` types is saturation; otherwise, overflow on `_Fract` types has undefined behavior.

When the state of the `FX_ACCUM_OVERFLOW` pragma is `SAT`, the overflow behaviour on `_Accum` types is saturation; otherwise, overflow on `_Accum` types has undefined behavior.

The default state for the `FX_FRACT_OVERFLOW` and `FX_ACCUM_OVERFLOW` pragmas is `DEFAULT`.

Note: the `DEFAULT` state of the overflow pragmas is intended to allow implementations to use the most optimal instruction sequences irrespective of their overflow behaviour for those computations where the actual overflow behaviour is not relevant; the actual overflow behaviour may be saturation, or anything else (including modular wrap-around) and may vary between different occurrences of the same operation, or even between different executions of the same operation.

If (after any overflow handling) the source value cannot be represented exactly by the fixed-point type, the source value is rounded to either the closest fixed-point value greater than the source value (rounded up) or to the closest fixed-point value less than the source value (rounded down).

For arithmetic operators other than `*` and `/`, the rounded result is returned as the result of the operation. The `*` and `/` operators may return either this rounded result or, depending of the state of the `FX_FULL_PRECISION` pragma, the closest larger or closest smaller value representable by the result fixed-point type. (Between rounding and this optional adjustment, the multiplication and division operations permit a mathematical error of almost 2 units in the last place of the result type.)

If the mathematical result of the `*` operator is exactly `1`, the closest smaller value representable by the fixed point result type may be returned as the result, even if the result type can represent the value `1` exactly. Correspondingly, if the mathematical result of the `*` operator is exactly `-1`, the closest larger value representable by the fixed point result type may be returned as the result, even if the result type can represent the value `-1` exactly. The circumstances in which a `1` or `-1` result might be replaced in this manner are implementation-defined.

Whether rounding is up or down is implementation-defined and may differ for different values and different situations.

Functions are supplied for arithmetic operations with different operand type and result type combinations (for instance, integer times fixed-point yielding integer, or integer divided by integer yielding a fixed-point type); see section 7.18.6.1.

### Section 6.2.5 - Types

add the following new paragraphs after paragraph 13.

There are six *fract* types, designated as **`short_Fract`**, **`_Fract`**, **`long_Fract`**, **`unsigned short_Fract`**, **`unsigned_Fract`**, and **`unsigned long_Fract`**. There are six *accum* types, designated as **`short_Accum`**, **`_Accum`**, **`long_Accum`**, **`unsigned short_Accum`**, **`unsigned_Accum`**, and **`unsigned long_Accum`**. The fract types and accum types are collectively called *fixed-point* types.

The minimum values for the number of fractional bits and the number of integral bits in the various fixed-point types are specified in section 5.2.4.2.3. An implementation may give any of the fixed-point types more fractional bits, and may also give any of the accum types more integral bits, subject to the following restrictions:

- Each unsigned fract type has either the same number of fractional bits or one more fractional bit than its corresponding signed fract type.
- Each fixed-point type is assigned a *fixed-point conversion rank*, as defined below. A fixed-point conversion rank is always greater than an integer conversion rank (see 6.3.1.1).
- When arranged in order of increasing rank, the number of fractional bits is nondecreasing for each of the following sets of fixed-point types:
  - signed fract types
  - unsigned fract types
  - signed accum types
  - unsigned accum types.
- When arranged in order of increasing rank, the number of integral bits is nondecreasing for each of the following sets of fixed-point types:
  - signed accum types
  - unsigned accum types
- Each signed accum type has at least as many integral bits as its corresponding unsigned accum type.

Furthermore, in order to promote consistency amongst implementations, the following are recommended practice where practical:

- The **`signed long_Fract`** type has at least 31 fractional bits.
- Each accum type has at least 8 integral bits.
- Each unsigned accum type has the same number of fractional bits as its corresponding unsigned fract type.
- Each signed accum type has the same number of fractional bits as either its corresponding signed fract type or its corresponding unsigned fract type.

**Section 6.2.5 - Types**, paragraph 17: change last sentence as follows.

Integer, fixed-point and real floating types are collectively called *real types*.

**Section 6.2.5 - Types**, paragraph 18: change first sentence as follows.

Integer, fixed-point and floating types are collectively called *arithmetic types*.

### **Section 6.2.6.3 - Fixed-point types (NEW SECTION)**

For unsigned fixed-point types, the bits of the object representation shall be divided into two groups: value bits and padding bits (there need not be any of the latter). There are two types of value bits: *integral* bits and *fractional* bits; if there are  $N$  value bits and  $L$  integral bits, then there are  $(N-L)$  fractional bits; for fract types, the number of integral bits is always zero ( $L=0$ ). For fract types, each bit shall represent a different power of 2 between  $2^{(-1)}$  and  $2^{(-N)}$ , so that objects of that type shall be capable of representing values from 0 to  $1-2^{(-N)}$  using a pure binary representation. For accum types, each bit shall represent a different power of 2 between  $2^{(L-1)}$  and  $2^{(L-N)}$ , so that objects of that type shall be capable of representing values from 0 to  $2^L-2^{(L-N)}$  using a pure binary representation. These representations shall be known as the value representations. The values of any padding bits are unspecified.

For signed fixed-point types, the bits of the object representation shall be divided into three groups: value bits, padding bits, and the sign bit. There need not be any padding bits; there shall be exactly one sign bit. There are two types of value bits: *integral* bits and *fractional* bits; if there are  $N$  value bits and  $L$  integral bits, then there are  $(N-L)$  fractional bits; for fract types, the number of integral bits is always zero ( $L=0$ ). For fract types, each bit shall represent a different power of 2 between  $2^{(-1)}$  and  $2^{(-N)}$ , so that objects of that type shall be capable of representing values from -1 to  $1-2^{(-N)}$  using a pure binary representation. For accum types, each bit shall represent a different power of 2 between  $2^{(L-1)}$  and  $2^{(L-N)}$ , so that objects of that type shall be capable of representing values from  $-2^L$  to  $2^L-2^{(L-N)}$  using a pure binary representation. These representations shall be known as the value representations. The values of any padding bits are unspecified.

The precision of a fixed-point type is the number of bits it uses to represent values, excluding any sign and padding bits. The width of a fixed-point type is the same but including any sign bit; thus for unsigned integer types the two values are the same, while for signed integer types the width is one greater than the precision.

### **Section 6.3.1.4 - Fixed-point types (NEW SECTION)**

The fixed-point types are assigned a *fixed-point rank*. The following types are listed in order of increasing rank:

`short _Fract, _Fract, long _Fract, short _Accum, _Accum, long _Accum`

Each unsigned fixed-point type has the same rank as its corresponding signed fixed-point type.

All conversions between a fixed-point type and another arithmetic type (which can be another fixed-point type) are defined. Overflow and rounding are handled according to the usual rules for the destination type. Conversions from a fixed-point to an integer type round toward zero. The rounding of conversions from a fixed-point type to a floating-point type is unspecified.

When a value of integer type or a value of fixed-point type is used in an expression with a value of fixed-point type, neither values are converted before the expression is evaluated.

EXAMPLE: in the following code fragment:

```
fract f = 0.1r;
int i = 3;

f = f * i;
```

the variable f gets the value 0.3.

#### Section 6.3.1.8 - Usual arithmetic conversions, after the conversion rule for conversion to **float**

Otherwise, if one operand has fixed-point type and the other operand has integer type, then no conversions are needed.

Otherwise, if both operands have signed fixed-point types, or if both operands have unsigned fixed-point types, then no conversions are needed.

Otherwise, if one operand has signed fixed-point type and the other operand has unsigned fixed-point type, the operand with unsigned type is converted to the signed fixed-point type corresponding to its own unsigned fixed-point type.

If the type of either of the operands has the **\_Sat** qualifier, the resulting type shall have the **\_Sat** qualifier.

#### Section 6.4.1 - Keywords, add the following new keywords:

**\_Accum**      **\_Fract**      **\_Sat**

#### Section 6.4.4.3 - Fixed-point constants (NEW SECTION)

##### Syntax

*fixed-constant*:

*decimal-fixed-constant*  
*hexadecimal-fixed-constant*

*decimal-fixed-constant*:

*fractional-constant exponent-part<sub>opt</sub>* *fixed-suffix*  
*digit-sequence exponent-part* *fixed-suffix*

*hexadecimal-fixed-constant*:

*hexadecimal-prefix* *hexadecimal-fractional-constant*  
*binary-exponent-part* *fixed-suffix*  
*hexadecimal-prefix* *hexadecimal-digit-sequence*  
*binary-exponent-part* *fixed-suffix*

*fixed-suffix*: *unsigned-suffix<sub>opt</sub>* *fpx-suffix<sub>opt</sub>* *fixed-qual*

*fpx-suffix*:

*long-suffix*  
*short-suffix*

*short-suffix*: one of

**h**    **H**

*fixed-qual*: one of

**k**    **K**    **r**    **R**

## Description

The description and semantics for a fixed-constant are the same as those for a floating constant with the following exceptions:

- fixed constants have always a suffix;
- fixed constant shall evaluate to a value that is in the range for the indicated type; an exception to this requirement is made for constants of a fract type with a value of exactly 1; such a constant shall denote the maximal value for the type.

**Section 6.5.3.3 - Unary arithmetic operators**, change second sentence of both paragraph 2 and 3 as follows:

If the type of the operand is an integer type, the integer promotions are performed on the operand, and the result has the promoted type; otherwise, the result has the same type as the operand type.

**Section 6.5.7 - Bitwise shift operands**, change the constraints section as follows:

The left operand shall have integer or fixed-point type, the right operand shall have integer type.

**Section 6.5.7 - Bitwise shift operands**, reword paragraphs 3-5 to indicate that these apply only to the full integer case, and add new paragraph 6:

If the left operand has a fixed-point type, the right operand is converted to **int** and must be nonnegative and less than the total number of (non-paddig) bits of the left operand. The type of the result is that of the left operand.

The result of **E1<<E2** is **E1\*2<sup>E2</sup>**, the result of **E1>>E2** is **E1\*2<sup>(-E2)</sup>**.

**Section 6.6 - Constant expressions**, change second sentence of paragraph 5 to start with

If a floating expression or a fixed-point expression is evaluated in the translation environment, ...

**Section 6.6 - Constant expressions**, change first sentence of paragraph 8 as follows:

An *arithmetic constant expression* shall have arithmetic type and shall only have operands that are integer constants, fixed-point constants, floating constants, enumeration constants, character constants, and **sizeof** expressions.

**Section 6.7.2 - Type specifiers**, add under **Syntax**, between **long** and **float**:

**\_Fract**  
**\_Accum**

**Section 6.7.2 - Type specifiers**, in paragraph 2 add before **float**:

- **signed short \_Fract**, or **short \_Fract**
- **signed \_Fract**, or **\_Fract**
- **signed long \_Fract**, or **long \_Fract**
- **signed short \_Accum**, or **short \_Accum**
- **signed \_Accum**, or **\_Accum**
- **signed long \_Accum**, or **long \_Accum**
- **unsigned short \_Fract**
- **unsigned \_Fract**
- **unsigned long \_Fract**
- **unsigned short \_Accum**
- **unsigned \_Accum**
- **unsigned long \_Accum**

**Section 6.7.3 - Type qualifiers**, add under **Syntax** after **volatile**:

\_Sat

**Section 6.7.3 - Type qualifiers**, add following sentences under **Constraints**:

Types other than fixed-point types shall not be sat-qualified.

**Section 6.7.3 - Type qualifiers**, add new paragraph 10:

When overflow occurs during the conversion of an arithmetic (source) value to a value of a sat-qualified target type, the value is replaced by the value from the target type that is the closest to the source value.

**Section 6.10.6 - Pragma directive**, add to the list in paragraph 2:

```
#pragma STDC FX_FULL_PRECISION on-off-switch  
#pragma STDC FX_FRACT_OVERFLOW overflow-switch  
#pragma STDC FX_ACCUM_OVERFLOW overflow-switch
```

*overflow-switch*: one of  
    SAT    DEFAULT

**Section 7.1.2 - Standard headers**, add to paragraph 2:

<stdfix.h>

**Section 7.18 Fixed-point arithmetic <stdfix.h> (NEW SECTION)**

**7.18.1 Introduction**

The header <stdfix.h> defines macros and declares functions that support fixed-point arithmetic. Each synopsis specifies a family of functions with, depending on the type of their parameters and return value, names with **r**, **k**, **h**, **l** or **u** prefixes or suffixes which are corresponding functions with fract type and accum type parameters or return values, with the optional type specifiers for **short**, **long** and **unsigned**.

The macro

**fract**

expands to **\_Fract**; the macro

**accum**

expands to `_Accum`.

The macro

`sat`

expands to `_Sat`.

Notwithstanding the provisions of 7.1.3, a program may undefine and perhaps then redefine the macros `fract`, `accum` and `sat`.

### 7.18.2 Integer types used as return types for the bits conversion functions

The following integer types are introduced as typedefs:

```
int_hr_t
int_uhr_t
int_r_t
int_ur_t
int_lr_t
int_ulr_t
int_hk_t
int_uhk_t
int_k_t
int_uk_t
int_lk_t
int_ulk_t
```

The type `int_fx_f` is the return type of the corresponding `bitsfx` function, and is chosen so that the return value can hold all the necessary bits. If there is no integer type available that is wide enough to hold the necessary bits for certain fixed-point types, the usage of the type is implementation defined.

### 7.18.3 Precision macros

New constants are introduced to denote the behavior and limits of fixed-point arithmetic.

A conforming implementation shall document all the limits specified in this section, as an addition to the limits required by the ISO C standard.

The values given below shall be replaced by constant expressions suitable for use in `#if` preprocessing directives.

The values in the following sections shall be replaced by constant expressions with implementation-defined values with the same type. Except for the various **EPSILON** values, their implementation-defined values shall be greater or equal in magnitude (absolute value) to those shown, with the same sign. For the various **EPSILON** values, their implementation-defined values shall be less or equal in magnitude to those shown.

- number of fractional bits for object of type **signed short \_Fract**  
**SFRACT\_FBIT** 7
- minimum value for an object of type **signed short \_Fract**  
**SFRACT\_MIN** (-0.5HR-0.5HR)
- maximum value for an object of type **signed short \_Fract**  
**SFRACT\_MAX** 0.9921875HR // decimal constant  
**SFRACT\_MAX** 0X1.FCP-1HR // hex constant
- the difference between 0.0HR and the least value greater than 0.0HR that is representable in the **signed short \_Fract** type  
**SFRACT\_EPSILON** 0.0078125HR // decimal constant  
**SFRACT\_EPSILON** 0X1P-7HR // hex constant
- number of fractional bits for object of type **unsigned short \_Fract**  
**USFRACT\_FBIT** 7
- maximum value for an object of type **unsigned short \_Fract**  
**USFRACT\_MAX** 0.9921875UHR // decimal constant  
**USFRACT\_MAX** 0X1.FCP-1UHR // hex constant
- the difference between 0.0UHR and the least value greater than 0.0UHR that is representable in the **unsigned short \_Fract** type  
**USFRACT\_EPSILON** 0.0078125UHR // decimal constant  
**USFRACT\_EPSILON** 0X1P-7UHR // hex constant
- number of fractional bits for object of type **signed \_Fract**  
**FRACT\_FBIT** 15
- minimum value for an object of type **signed \_Fract**

**FRACT\_MIN (-0.5R-0.5R)**

- maximum value for an object of type **signed \_Fract**

```
FRACT_MAX 0.999969482421875R           // decimal constant
FRACT_MAX 0X1.FFFCP-1R                   // hex constant
```

- the difference between 0.0R and the least value greater than 0.0R that is representable in the **signed \_Fract** type

```
FRACT_EPSILON 0.000030517578125R        // decimal constant
FRACT_EPSILON 0X1P-15R                   // hex constant
```

- number of fractional bits for object of type **unsigned \_Fract**

**UFRACT\_FBIT 15**

- maximum value for an object of type **unsigned \_Fract**

```
UFRACT_MAX 0.999969482421875UR          // decimal constant
UFRACT_MAX 0X1.FFFCP-1UR                  // hex constant
```

- the difference between 0.0UR and the least value greater than 0.0UR that is representable in the **unsigned \_Fract** type

```
UFRACT_EPSILON 0.000030517578125UR       // decimal constant
UFRACT_EPSILON 0X1P-15UR                  // hex constant
```

- number of fractional bits for object of type **signed long \_Fract**

**LFRACT\_FBIT 23**

- minimum value for an object of type **signed long \_Fract**

**LFRACT\_MIN (-0.5LR-0.5LR)**

- maximum value for an object of type **signed long \_Fract**

```
LFRACT_MAX 0.99999988079071044921875LR
           // decimal constant
LFRACT_MAX 0X1.FFFFFCP-1LR                // hex constant
```

- the difference between 0.0LR and the least value greater than 0.0LR that is representable in the **signed long \_Fract** type

- ```
LFRACT_EPSILON 0.00000011920928955078125LR          // decimal constant
LFRACT_EPSILON 0X1P-23LR                            // hex constant

- number of fractional bits for object of type unsigned long _Fract

ULFRACT_FBIT 23

- maximum value for an object of type unsigned long _Fract

ULFRACT_MAX 0.99999988079071044921875ULR          // decimal constant
ULFRACT_MAX 0X1.FFFFFCP-1ULR                        // hex constant

- the difference between 0.0ULR and the least value greater than 0.0ULR that is representable
in the unsigned long _Fract type

ULFRACT_EPSILON 0.00000011920928955078125ULR          // decimal constant
ULFRACT_EPSILON 0X1P-23ULR                          // hex constant

- number of fractional bits for object of type signed short _Accum

SACCUM_FBIT 7

- number of integral bits for object of type signed short _Accum

SACCUM_IBIT 4

- minimum value for an object of type signed short _Accum

SACCUM_MIN (-8.0HK-8.0HK)

- maximum value for an object of type signed short _Accum

SACCUM_MAX 15.9921875HK                           // decimal constant
SACCUM_MAX 0X1.FFCP+3HK                           // hex constant

- the difference between 0.0HK and the least value greater than 0.0HK that is representable in
the signed short _Accum type

SACCUM_EPSILON 0.0078125HK                         // decimal constant
SACCUM_EPSILON 0X1P-7HK                           // hex constant

- maximum value for an object of type unsigned short _Accum
```

```
USACCUM_MAX 15.9921875UHK           // decimal constant
USACCUM_MAX 0X1.FFCP+3UHK          // hex constant
```

- the difference between 0.0UHK and the least value greater than 0.0UHK that is representable in the **unsigned short \_Accum** type

```
USACCUM_EPSILON 0.0078125UHK        // decimal constant
USACCUM_EPSILON 0X1P-7UHK          // hex constant
```

- number fractional of bits for object of type **signed \_Accum**

```
ACCUM_FBIT 15
```

- number of integral bits for object of type **signed \_Accum**

```
ACCUM_IBIT 4
```

- minimum value for an object of type **signed \_Accum**

```
ACCUM_MIN (-8.0K-8.0K)
```

- maximum value for an object of type **signed \_Accum**

```
ACCUM_MAX 15.999969482421875K      // decimal constant
ACCUM_MAX 0X1.FFFFCP+3K            // hex constant
```

- the difference between 0.0K and the least value greater than 0.0K that is representable in the **signed \_Accum** type

```
ACCUM_EPSILON 0.000030517578125K    // decimal constant
ACCUM_EPSILON 0X1P-15K              // hex constant
```

- maximum value for an object of type **unsigned \_Accum**

```
UACCUM_MAX 15.999969482421875UK    // decimal constant
UACCUM_MAX 0X1.FFFFCP+3UK          // hex constant
```

- the difference between 0.0UK and the least value greater than 0.0UK that is representable in the **unsigned \_Accum** type

```
UACCUM_EPSILON 0.000030517578125UK  // decimal constant
UACCUM_EPSILON 0X1P-15UK            // hex constant
```

- number of fractional bits for object of type **signed long \_Accum**

- **LACCUM\_FBIT** 23
  - number of integral bits for object of type **signed long \_Accum**
  
- **LACCUM\_FBIT** 4
  - minimum value for an object of type **signed long \_Accum**
  
- **LACCUM\_MIN** (-8.0LK-8.0LK)
  - maximum value for an object of type **signed long \_Accum**
  
- **LACCUM\_MAX** 15.9999988079071044921875LK
  - decimal constant
- **LACCUM\_MAX** 0X1.FFFFFFFCP+3LK
  - hex constant
  
- the difference between 0.0LK and the least value greater than 0.0LK that is representable in the **signed long \_Accum** type
  
- **LACCUM\_EPSILON** 0.00000011920928955078125LK
  - decimal constant
- **LACCUM\_EPSILON** 0X1P-23LK
  - hex constant
  
- maximum value for an object of type **unsigned long \_Accum**
  
- **ULACCUM\_MAX** 15.9999988079071044921875ULK
  - decimal constant
- **ULACCUM\_MAX** 0X1.FFFFFFFCP+3ULK
  - hex constant
  
- the difference between 0.0ULK and the least value greater than 0.0ULK that is representable in the **unsigned long \_Accum** type
  
- **ULACCUM\_EPSILON** 0.00000011920928955078125ULK
  - decimal constant
- **ULACCUM\_EPSILON** 0X1P-23ULK
  - hex constant

#### 7.18.4 The **FX\_FULL\_PRECISION** pragma

##### Synopsis

```
#include <stdfix.h>
#pragma STDC FX_FULL_PRECISION on-off-switch
```

##### Description

The normal required precision for fixed-point operations is 1 ulp. However, in certain environments a precision of 2 ulps on multiply and divide operations is enough, and such relaxed requirements may result in a significantly increased execution speed. The **FX\_FULL\_PRECISION** pragma can be used to inform the implementation that (where the state is "off") the relaxed requirements are allowed. If the indicated state is "on", the implementation is required to return results with full precision. Each pragma can occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another **FP\_FULL\_PRECISION** pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another **FP\_FULL\_PRECISION** pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined. The default state ("on" or "off") for the pragma is implementation defined.

### 7.18.5 The fixed-point overflow pragmas (**FX\_FRACT\_OVERFLOW**, **FX\_ACCUM\_OVERFLOW**)

#### Synopsis

```
#include <stdfix.h>
#pragma STDC FX_FRACT_OVERFLOW overflow-switch
#pragma STDC FX_ACCUM_OVERFLOW overflow-switch

overflow-switch: one of
    SAT   DEFAULT
```

#### Description

When neither operand of a fixed-point operator have the **\_Sat** type qualifier, the overflow behavior is controlled by either the **FX\_FRACT\_OVERFLOW** or the **FX\_ACCUM\_OVERFLOW** pragma, depending on the type of the overflowing value. When the state of an overflow pragma is **DEFAULT**, fixed-point overflow for the cooresponding type has undefined behavior. Otherwise, the overflow behaviour is according to the set state. The default state of the overflow pragmas is **DEFAULT**. Each pragma can occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another overflow pragma is encountered, or until the end of the translation unit. When inside a compound statement, a pragma takes effect from its occurrence until another occurrence of the same pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined.

### 7.18.6 The fixed-point intrinsic functions

### 7.18.6.1 The fixed-point arithmetic operation support functions

#### Synopsis

```

        int mulir (      int,      fract );
long int mulilr( long int, long fract );
        int mulik (      int,      accum );
long int mulilk( long int, long accum );

        int divir (      int,      fract );
long int divilr( long int, long fract );
        int divik (      int,      accum );
long int divilk( long int, long accum );

        fract rdivi (      int,      int );
long fract lrdivi( long int, long int );
        accum kdivi (      int,      int );
long accum lkdivi( long int, long int );

        int idivr (      fract,      fract );
long int idivlr( long fract, long fract );
        int idivk (      accum,      accum );
long int idivlk( long accum, long accum );

```

#### Description

The above functions compute the mathematically exact result of the multiplication or division operation on the operands with the indicated types, and return a value with the indicated type. The unsigned versions of the above functions (**muluir**, **mulilr**, **muliuk**, **mululk**, **diviur**, **diviulr**, **diviuk**, **diviulk**, **urdivi**, **ulrdivi**, **ukdivi**, **ulkdivi**, **idivur**, **idivulr**, **idivuk** and **idivulk** respectively) take values of the two corresponding unsigned types, and return a value of the corresponding unsigned type.

#### Returns

For functions returning an integer value, the return value is rounded towards zero. For functions returning a fixed-point value, the return value is saturated. If the second operand of one of the divide functions is zero, the behaviour is undefined.

### 7.18.6.2 The fixed-point absolute value functions

#### Synopsis

```
#include <stdfix.h>
short fract abshr(short fract f);
```

```

fract absr(fract f);
long fract abslr(long fract f);
short accum abshk(short accum f);
accum absk(accum f);
long accum abslk(long accum f);

```

**Description**

The above functions compute the absolute value of a fixed-point value *f*.

**Returns**

The functions return  $|f|$ . If the exact result cannot be represented, the saturated result is returned.

**7.18.6.3 The fixed-point round functions****Synopsis**

```

#include <stdfix.h>
short fract roundhr(short fract f, int n);
fract roundr(fract f, int n);
long fract roundlr(long fract f, int n);
short accum roundhk(short accum f, int n);
accum roundk(accum f, int n);
long accum roundlk(long accum f, int n);
unsigned short fract rounduhr(unsigned short fract f, int n);
unsigned fract roundur(unsigned fract f, int n);
unsigned long fract roundulr(unsigned long fract f, int n);
unsigned short accum rounduhk(unsigned short accum f, int n);
unsigned accum rounduk(unsigned accum f, int n);
unsigned long accum roundulk(unsigned long accum f, int n);

```

**Description**

The above functions compute the value of *f*, rounded to the number of fractional bits specified in *n*. The rounding applied is to-nearest, with unspecified rounding direction in the halfway case. Fractional bits beyond the rounding point are set to zero in the result. The value of *n* must be nonnegative and less than the number of fractional bits in the fixed-point type of *f*.

**Returns**

The functions return the rounded result, as specified. If the value of *n* is negative or larger than the number of fractional bits in the fixed-point type of *f*, the result is unspecified. If the exact result cannot be represented, the saturated result is returned.

**7.18.6.4 The fixed-point countls functions**

## Synopsis

```
#include <stdfix.h>
int countlshr(short fract f);
int countlsr(fract f);
int countlslr(long fract f);
int countlshk(short accum f);
int countlsk(accum f);
int countlslk(long accum f);
int countlsuhr(unsigned short fract f);
int countlsur(unsigned fract f);
int countlsulr(unsigned long fract f);
int countlsuhk(unsigned short accum f);
int countlsuk(unsigned accum f);
int countlslk(unsigned long accum f);
```

## Description

The integer return value of the above functions is defined as follows:

- if the value of the fixed-point argument **f** is non-zero, the return value is the largest integer **k** for which the expression **f<<k** does not overflow;
- if the value of the fixed-point argument is zero, an integer value is returned that is at least as large as N-1, where N is the total number of (nonpadding) bits of the fixed-point type of the argument.

Note: if the value of the fixed-point argument is zero, the recommended return value is exactly N-1.

## Returns

The **countls** functions return the integer value as indicated.

### 7.18.6.5 The bitwise fixed-point to integer conversion functions

## Synopsis

```
#include <stdfix.h>
int_hr_t bitshr(short fract f);
int_r_t bitsr(fract f);
int_lr_t bitslr(long fract f);
int_hk_t bitshk(short accum f);
int_k_t bitsk(accum f);
int_lk_t bitslk(long accum f);
int_uhr_t bitsuhr(unsigned short fract f);
int_ur_t bitsur(unsigned fract f);
int_ulr_t bitsulr(unsigned long fract f);
```

```
int_uhk_t bitsuhk(unsigned short accum f);
int_uk_t bitsuk(unsigned accum f);
int_ulk_t bitsulk(unsigned long accum f);
```

## Description

The **bits** functions return an integer value equal to the fixed-point value of **f** multiplied by  $2^F$ , where F is the number of fractional bits in the type of **f**. The result type is an integer type big enough to hold all valid result values for the given fixed-point argument type. For example, if the **fract** type has 15 fractional bits, then after the declaration

```
fract a = 0.5;
```

the value of **bitsr(a)** is  $0.5 * 2^{15} = 0x4000$ .

## Returns

The **bits** functions return the value of the argument as an integer bitpattern as indicated.

### 7.18.6.6 The bitwise integer to fixed-point conversion functions

## Synopsis

```
#include <stdfix.h>
short fract hrbits(int n);
fract rbits(int n);
long fract lrbits(int n);
short accum hkbits(int n);
accum kbits(int n);
long accum lkbits(int n);
unsigned short fract uhrbits(unsigned int n);
unsigned fract urbits(unsigned int n);
unsigned long fract ulrbits(unsigned int n);
unsigned short accum uhkbits(unsigned int n);
unsigned accum ukbits(unsigned int n);
unsigned long accum ulkbits(unsigned int n);
```

## Description

The above functions return an fixed-point value equal to the integer value of the argument divided by  $2^F$ , where F is the number of fractional bits in the fixed-point result type of the function. For example, if **fract** has 15 fractional bits, then the value of **rbits(0x2000)** is 0.25.

## Returns

The above functions return the indicated value.

### 7.18.6.7 Type-generic fixed-point functions

For each of the fixed-point absolute value functions in 7.18.6.2, the fixed-point round functions in 7.18.6.3 and the fixed-point countls functions in 7.18.6.4, a type-generic macro is defined as follows:

|                                          | <u>type-generic macro</u> |
|------------------------------------------|---------------------------|
| the fixed-point absolute value functions | <b>absfx</b>              |
| the fixed-point round functions          | <b>roundfx</b>            |
| the fixed-point countls functions        | <b>countlsfx</b>          |

### 7.18.6.8 Numeric conversion functions

#### Synopsis

```
#include <stdfix.h>
short fract strtohr(const char * restrict nptr,
                     char ** restrict endptr);
fract strtor(const char * restrict nptr,
              char ** restrict endptr);
long fract strtolr(const char * restrict nptr,
                    char ** restrict endptr);
short accum strtohk(const char * restrict nptr,
                     char ** restrict endptr);
accum strtok(const char * restrict nptr,
              char ** restrict endptr);
long accum strtolk(const char * restrict nptr,
                    char ** restrict endptr);
unsigned short fract strtouhr(const char * restrict nptr,
                               char ** restrict endptr);
unsigned fract strtour(const char * restrict nptr,
                       char ** restrict endptr);
unsigned long fract strtoulr(const char * restrict nptr,
                            char ** restrict endptr);
unsigned short accum strtouhk(const char * restrict nptr,
                               char ** restrict endptr);
unsigned accum strtouk(const char * restrict nptr,
                       char ** restrict endptr);
unsigned long accum strtoulk(const char * restrict nptr,
                            char ** restrict endptr);
```

#### Description

The **strtohr**, **strtor**, **strtolr**, **strtohk**, **strtok**, **strtolk**, **strtouhr**, **strtour**, **strtoulr**, **strtouhk**, **strtouk** and **strtoulk** functions convert the initial portion of the string pointed to by **nptr** to **short fract**, **fract**, **long fract**, **short accum**, **accum**, **long accum**, **unsigned short fract**, **unsigned fract**, **unsigned long fract**, **unsigned short accum**, **unsigned accum**, and **unsigned long accum** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the **isspace** function), a subject sequence resembling a fixed-point constant; and a final string of one or more unrecognised characters, including the terminating null character of the input string. Then, they attempt to convert the subject sequence to a fixed-point number, and return the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part as defined in 6.4.4.3;
- a **0x** or **0X**, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point character, then an optional binary exponent part as defined in 6.4.4.3.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

If the subject sequence has the expected form for a fixed-point number, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as a fixed-point constant according to the rules of 6.4.4.3, except that the decimal-point character is used in place of a period, and that if neither an exponent part nor a decimal-point character appears in a decimal fixed-point number, or if a binary exponent part does not appear in a hexadecimal fixed-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence is interpreted as negated. A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

The value resulting from the conversion is correctly rounded.

In other than the "**C**" locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

## Returns

The functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, a saturated result is

returned (according to the return type and sign of the value), and the value of the macro ERANGE is stored in errno.

**Section 7.19.6.1 - The fprintf function**, paragraph 4, third bullet, change begin of first sentence to  
An optional *precision* that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal-point character for **a**, **A**, **e**, **E**, **f**, **F**, **r**, **R**, **k** and **K** conversions, . . .

**Section 7.19.6.1 - The fprintf function**, paragraph 6, the '#' bullet, change begin of fourth sentence to

For **a**, **A**, **e**, **E**, **f**, **F**, **g**, **G**, **r**, **R**, **k** and **K** conversions, . . .

**Section 7.19.6.1 - The fprintf function**, paragraph 6, the 'o' bullet, change begin of first sentence to

For **d**, **i**, **o**, **u**, **x**, **X**, **a**, **A**, **e**, **E**, **f**, **F**, **g**, **G**, **r**, **R**, **k** and **K** conversions, . . .

**Section 7.19.6.1 - The fprintf function**, paragraph 7, the 'h' bullet, add at the end of the first sentence:

that a following **r**, **R**, **k** or **K** conversion specifier applies to a short fixed-point type argument.

**Section 7.19.6.1 - The fprintf function**, paragraph 7, the 'l (ell)' bullet, add before last semi-colon:

that a following **r**, **R**, **k** or **K** conversion specifier applies to a fixed-point type argument;

**Section 7.19.6.1 - The fprintf function**, paragraph 8, add new bullet before the 'c' bullet:

**r**, **R**, **k**, **K**      A signed fixed-point fract type (**r**), an unsigned fract type (**R**), a signed accum type (**k**) or an unsigned accum type (**K**) representing a fixed-point number is converted to decimal notation in the style **[-]ddd.ddd**, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

**Section 7.19.6.1 - The fprintf function**, paragraph 13, change beginning of first sentence to

For **e**, **E**, **f**, **F**, **g**, **G**, **r**, **R**, **k** and **K** conversions, . . .

**Section 7.19.6.2 - The fscanf function**, paragraph 11, the 'h' bullet, add at the end of the first sentence:

that a following **r**, **R**, **k** or **K** conversion specifier applies to an argument with type pointer to short fixed-point type.

**Section 7.19.6.2 - The fscanf function**, paragraph 11, the '1 (ell)' bullet, insert after last semicolon:

that a following **r**, **R**, **k** or **K** conversion specifier applies to an argument with type pointer to long fixed-point type;

**Section 7.19.6.2 - The fscanf function**, paragraph 12, add new bullet before the 'c' bullet:

**r, R, k, K**      Matches an optionally signed fixed-point number, whose format is the same as expected for the subject sequence of the **strtodfx** functions. The corresponding argument shall be a pointer to a signed fract type (**r**), a pointer to an unsigned fract type (**R**), a pointer to a signed accum type (**k**) or a pointer to an unsigned fract type (**K**).

### 3 Named address spaces and named-register storage classes

#### 3.1 Overview and principles of named address spaces

##### 3.1.1 Additional address spaces

Embedded applications are often implemented on processors with multiple independent address spaces. An embedded processor may have separate address spaces, for example, for a general-purpose memory, for smaller, faster memories, for code and constant data, and for I/O accesses. Typically, the distinction between various address spaces is built into the processor's instruction set, with accesses to different spaces requiring distinctly different instructions. For more discussion on address spaces, see Annex B of this Technical Report.

ISO/IEC 9899:1999 stipulates that all objects are allocated in a single, common address space. For the purpose of this Technical Report, the C language is extended to support additional address spaces. When not specified otherwise, objects are allocated by default in a *generic* address space, which corresponds to the single address space of ISO/IEC 9899:1999. In addition to the generic address space, an implementation may support other, *named* address spaces. Objects may be allocated in these alternate address spaces, and pointers may be defined that point to objects in these address spaces. It is intended that pointers into an address space only need be large enough to support the range of addresses in that address space.

The named address space approach, as described in this section, enables application portability between systems with similar memory configurations. This portability is at source code level, and not at binary or execution level.

##### 3.1.2 Address-space type qualifiers

Each address space other than the generic one has a unique name in the form of an identifier. Address space names are ordinary identifiers, sharing the same name space as variables and `typedef` names. Any such names follow the same rules for scope as other ordinary identifiers (such as `typedef` names). An implementation may provide an implementation-defined set of *intrinsic* address spaces that are, in effect, pre-defined at the start of every translation unit. The names of intrinsic address spaces must be reserved identifiers (beginning with an underscore and an uppercase letter or with two underscores). An implementation may also optionally support a means for new address space names to be defined within a translation unit.

An address space name can be used in a declaration to specify the address space in which an object will be allocated. The C syntax for type qualifiers is extended to include an address space name as a valid type qualifier. If the type of an object is qualified by an address space name, the object is allocated in the specified address space; otherwise, the object is allocated in the generic address space. Note that, since a function is not an object, functions are always allocated in the generic address space.

Example:

If an implementation provides intrinsic address spaces with names `_X` and `_Y`, the following are valid declarations:

```
_X char a, b, c;
    // Declares three characters in address space _X

_X const int *p;
    // Declares a pointer in the generic address space
    // that points to a constant int object in
    // address space _X

_X struct { int a; char b; } *_Y q;
    // Declares a pointer in address space _Y that points
    // to a structure in address space _X
```

There are some constraints on the use of address-space type qualifiers. The most significant constraint is that an address space name cannot be used to qualify an object that has automatic storage duration. This implies, in particular, that variables declared inside a function cannot be allocated in a named address space unless they are also explicitly declared as `static` or `extern`.

For the examples above, the declarations of `a`, `b`, `c`, and `q` would have to be outside any function to be valid. The declaration of `p` would be acceptable within a function, because `p` itself would be in the generic address space.

Other limitations are detailed in section 3.3 below.

### 3.1.3 Address space nesting and rules for pointers

Address spaces may overlap in a nested fashion. For any two address spaces, either the address spaces must be disjoint, they must be equivalent, or one must be a subset of the other. Other forms of overlapping are not permitted. If an object is in address space **A** and **A** is a subset of address space **B**, the object is simultaneously in address space **B**.

An implementation must define the relationship between all pairs of address spaces. (The complete set of address spaces includes the generic address space plus any address spaces that may be defined within a translation unit, if the implementation supports such definitions within a program.) There is no requirement that named address spaces (intrinsic or otherwise) be subsets of the generic address space.

As determined by its type, every pointer points into a specific address space, either the generic address space or a named address space. A pointer into an address space can only point to locations in that address space (including any subset address spaces).

A non-null pointer into an address space **A** can be cast to a pointer into another address space **B**, but such a cast is undefined if the source pointer does not point to a location in **B**. Note that if **A** is a subset of **B**, the cast is always valid; however, if **B** is a subset of **A**, the cast is valid only if the source pointer refers to a location in **B**. A null pointer into one address space can be cast to any pointer into an overlapping address space.

If a pointer into address space **A** is assigned to a pointer into a different address space **B**, a constraint requires that **A** be a subset of **B**. (As usual, this constraint can be circumvented by a cast of the source pointer before the assignment.)

### 3.1.4 Standard library support

The standard C library (ISO/IEC 9899:1999 section 7 - Libraries) is unchanged; the library's functions and objects continue to be declared only with regard to the generic address space. One consequence is that pointers into named address spaces cannot be passed as arguments to library functions except in the special case that the named address spaces are subsets of the generic address space. Likewise, library functions such as `malloc` that allocate memory do so only for the generic address space; there are no standard functions for allocating space in other address spaces.

## 3.2 Overview and principles of named-register storage classes

### 3.2.1 Access to machine registers

Embedded applications sometimes need to access processor registers that are not addressable in any of the machine's address spaces. The reasons for accessing such a register might include:

- the application must manipulate the register to achieve certain side effects;
- memory is at a premium and the register is needed as a storage location; and/or
- other software (e.g., an operating system) expects certain information to be stored in the register.

For the purpose of this Technical Report, the C language is extended to permit the allocation of variables within named registers. Allocating a variable in a named register essentially establishes the variable as an alias for that register.

### 3.2.2 Named-register storage-class specifiers

Just like named address spaces, each of the register names supported by an implementation has a unique name in the form of an identifier. Register names are ordinary identifiers, sharing the same name space as variables and `typedef` names. Any such names follow the same rules for scope as other ordinary identifiers (just like `typedef` names and address space names). An implementation

may provide an implementation-defined set of *intrinsic* register names that are, in effect, pre-defined at the start of every translation unit. The names of intrinsic registers must be reserved identifiers (beginning with an underscore and an uppercase letter or with two underscores). An implementation may also optionally support a means for new register names to be defined within a translation unit.

The C syntax for storage-class specifiers is extended to include the sequence

```
register register-name
```

(i.e., the **register** keyword followed by a register name identifier) as a valid storage-class specifier. Such a storage-class specifier is a *named-register storage-class specifier*, not to be confused with a **register** storage-class specifier which ISO/IEC 9899:1999 defines as being the **register** keyword alone.

A named-register storage-class specifier is permitted only for identifiers designating an object. Like the plain register storage class, the address-of operator & cannot be used to take the address of an identifier declared with a named-register storage class. However, in contrast to plain **register**, an identifier declared with a named-register storage class has external linkage and static storage duration. Thus, named-register storage classes operate more like **extern** than plain **register**.

As there is only one instance of any machine register, a program may not declare more than one identifier with the same named-register storage class. Furthermore, the type of an object declared with a named-register storage class cannot be an array and must not be of a size larger than the register itself. If the type of the object is smaller than the register, the portion of the register that actually corresponds to the declared identifier is implementation-defined.

Examples:

Assuming `_DP` and `_CC` are the names of char-sized intrinsic registers, the following are possible valid declarations:

```
register _DP volatile unsigned char direct_page_reg;

register _CC volatile struct {
    int is_IRQ : 1;
    int disable_FIRQ : 1;
    int half_carry : 1;
    int disable_IRQ : 1;
    int negative : 1;
    int zero : 1;
    int overflow : 1;
    int carry : 1;
} cond_reg;
```

However, a declaration associating a different identifier with one of the same registers is not permitted:

```
register _DP volatile unsigned char DP_reg;
// Not allowed; conflicts with previous declaration of
// direct_page_reg.
```

### 3.2.3 Ensuring correct side effects via objects allocated in registers

If reading from or writing to a register has a side effect in the machine (as is often the case for I/O registers), an object allocated in a named register may need to be declared **volatile** to ensure that accesses to the object (and hence to the register) occur as they appear in the C source code. Even then, it is important to be aware that ISO/IEC 9899:1999 (Section 6.7.3, paragraph 6) stipulates

What constitutes an access to an object that has volatile-qualified type is implementation-defined.

Thus, in theory at least, achieving a specific sequence of reads from and writes to a physical register depends on implementation-defined behavior, even if the C object allocated in the register is declared **volatile**. In practice, to avoid this problem, the C code will likely need to be written in a conservative style, e.g., without ambiguous expressions.

### 3.2.4 Relationship between named registers and I/O-register designators

Whether register names as defined above are also valid I/O-register designators as defined in Section 4 is implementation-defined. An implementation may or may not permit storage-class register names to be used as I/O-register designators in the macros of Section 4 such as **iord** and **iowr**. In particular, although an implementation must support the basic I/O hardware addressing of Section 4, it can at the same time not provide any intrinsic named registers or any means by which other named registers can be declared in a program.

## 3.3 Impact on the C language usage.

This section details the modifications to ISO/IEC 9899:1999 needed to incorporate the functionality of named address spaces and named-register storage classes overviewed in Sections 3.1 and 3.2 of this Technical Report. The changes listed in this section are limited to syntax and semantics; examples, forward references, and other descriptive information are omitted. The modifications are ordered according to the sections of ISO/IEC 9899:1999 to which they refer. If a section of ISO/IEC 9899:1999 is not mentioned, no changes to that section are needed. New sections are indicated with **(NEW SECTION)**.

**Section 6.2.1 - Scopes of identifiers**, change the first sentence of paragraph 1 to:

An identifier can denote an object; a function; an address space; a named register; a tag or a member of a structure, union, or enumeration; a typedef name; a label name; a macro name; or a macro parameter.

**Section 6.2.1 - Scopes of identifiers**, replace the first two sentences of paragraph 4 with:

Every other identifier has scope determined by the placement of its declaration (in a declarator or type specifier), if any. (The names of intrinsic address spaces and intrinsic registers are implicitly declared as specified in 6.2.4a and 6.7.1.1 below.) If the identifier is implicitly declared, or if the declarator or type specifier that declares the identifier appears outside of any block or list of parameters, the identifier has *file scope*, which terminates at the end of the translation unit.

**Section 6.2.1 - Scopes of identifiers**, add at the start of paragraph 7:

Identifiers that are implicitly declared have scope that begins at the start of a translation unit.

**Section 6.2.2 - Linkages of identifiers**, add a new paragraph between existing paragraphs 3 and 4:

If the declaration of an identifier contains a named-register storage-class specifier, the identifier has external linkage.

**Section 6.2.2 - Linkages of identifiers**, change paragraph 6 to:

The following identifiers have no linkage: an identifier declared to be anything other than an object or a function; an identifier declared to be a function parameter; a block scope identifier for an object declared without either a named-register storage-class specifier or the storage-class specifier **extern**.

**Section 6.2.4a - Address spaces (NEW SECTION)**

Objects are allocated in one or more *address spaces*. A unique *generic address space* always exists. Every address space other than the generic one has a unique name in the form of an identifier. Address spaces other than the generic one are called *named address spaces*. An object is always completely allocated into at least one address space.

Some (possibly empty) implementation-defined set of named address spaces are *intrinsic*. The name of an intrinsic address space shall begin with an underscore and an uppercase letter or with two underscores (and hence is a reserved identifier as defined in 7.1.3). There is no declaration for the name of an intrinsic address space in a translation unit; the identifier is *implicitly declared* with a scope covering the entire translation unit.

An implementation may optionally support an implementation-defined syntax for declaring other (not intrinsic) named address spaces.

Each address space (intrinsic or otherwise) exists for the entire execution of the program.

If address space **A** encloses address space **B**, then every location (address) within **B** is also within **A**. (Either **A** or **B** may be the generic address space.) The property of “enclosing” is transitive: if **A** encloses **B** and **B** encloses a third address space **C**, then **A** also encloses **C**. Every address space encloses itself.

For every pair of distinct address spaces **A** and **B**, it is implementation-defined whether **A** encloses **B**.

If one address space encloses another, the two address spaces *overlap*, and their *combined* address space is the one that encloses the other. If two address spaces do not overlap, they are *disjoint*, and no location (address) within one is also within the other. (Thus if two address spaces share a location, one address space must enclose the other.)

**Section 6.2.5 - Types**, replace the second sentence of paragraph 25 with:

Each unqualified type has several *qualified versions* of its type,<sup>38)</sup> corresponding to the combinations of one, two, or all three of the **const**, **volatile**, and **restrict** qualifiers, and all combinations of any subset of these three qualifiers with one address space qualifier. (Syntactically, an address space qualifier is an address space name, so there is an address space qualifier for each visible address space name.)

**Section 6.2.5 - Types**, replace paragraph 26 with three paragraphs:

The qualifiers **const**, **volatile**, and **restrict** are *access qualifiers*. An *additionally access-qualified version* of a type is the same type with any (or no) additional access qualifiers.

If type **T** is qualified by the address space qualifier for address space **A**, then “**T** is in **A**”. If type **T** is not qualified by an address space qualifier, then **T** is in the generic address space. If type **T** is in address space **A**, a pointer to **T** is also a “pointer into **A**”, and the *referenced address space* of the pointer is **A**.

A pointer to **void** in any address space shall have the same representation and alignment requirements as a pointer to a character type in the same address space.<sup>39)</sup> Similarly, pointers to differently access-qualified versions of compatible types shall have the same representation and alignment requirements. All pointers to structure types in the same address space shall have the same representation and alignment requirements as each other. All pointers to union types in the same address space shall have the same representation and alignment requirements as each other. Pointers to other types need not have the same representation or alignment requirements.

**Section 6.3.2.3 - Pointers**, replace the first two paragraphs with three paragraphs:

If, as provided below, a pointer into one address space is converted to a pointer into another address space, then unless the original pointer is a null pointer (defined below) or the location referred to by the original pointer is within the second address space, the behavior is undefined. (For the original pointer to refer to a location within the second address space, the two address spaces must overlap.)

A pointer to **void** in any address space may be converted to or from a pointer to any incomplete or object type. A pointer to any incomplete or object type in some address space may be converted to a pointer to **void** in an enclosing address space and back again; the result shall compare equal to the original pointer.

A pointer to a type may be converted to a pointer to a differently access-qualified version of the type (but with the same address-space qualifier, if any); the original and converted pointers shall compare equal.

**Section 6.3.2.3 - Pointers**, replace the last sentence of paragraph 4 with:

If the referenced address spaces of the original and converted pointers are disjoint, the behavior is undefined. Any two null pointers whose referenced address spaces overlap shall compare equal.

**Section 6.5 - Expressions**, replace the first four bullets of paragraph 7 with two bullets:

- an additionally access-qualified version of a type compatible with the effective type of the object,
- a type that is the signed or unsigned type corresponding to an additionally access-qualified version of the effective type of the object,

**Section 6.5.2.2 - Function calls**, change the second bullet of paragraph 6 to:

- both types are pointers to unqualified or access-qualified versions of a character type or **void** in the same address space.

**Section 6.5.2.5 - Compound literals**, add another constraint paragraph:

If the compound literal occurs inside the body of a function, the type name shall not be qualified by an address-space qualifier.

**Section 6.5.3.2 - Address and indirection operators**, change the end of paragraph 1 to:

... and is not declared with the **register** storage-class specifier or with a named-register storage-class specifier.

**Section 6.5.4 - Cast operators**, add another constraint paragraph:

If the type-name is qualified by an address-space qualifier, the type-name shall specify a pointer type.

**Section 6.5.8 - Relational operators**, add another constraint paragraph:

If the two operands are pointers into different address spaces, the address spaces must overlap.

**Section 6.5.8 - Relational operators**, add a new paragraph between existing paragraphs 3 and 4:

If the two operands are pointers into different address spaces, one of the address spaces encloses the other. The pointer into the enclosed address space is first converted to a pointer to the same reference type except with any address-space qualifier removed and any address-space qualifier of the other pointer's reference type added. (After this conversion, both operands are pointers into the same address space.)

**Section 6.5.9 - Equality operators**, add another constraint paragraph:

If the two operands are pointers into different address spaces, the address spaces must overlap.

**Section 6.5.9 - Equality operators**, replace paragraph 5 with:

Otherwise, at least one operand is a pointer. If one operand is a pointer and the other is a null pointer constant, the null pointer constant is converted to the type of the pointer. If both operands are pointers, each of the following conversions is performed as applicable:

- If the two operands are pointers into different address spaces, one of the address spaces encloses the other. The pointer into the enclosed address space is first converted to a pointer to the same reference type except with any address-space qualifier removed and any address-space qualifier of the other pointer's reference type added. (After this conversion, both operands are pointers into the same address space.)
- Then, if one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **void**, the former is converted to the type of the latter.

**Section 6.5.15 - Conditional operator**, add another constraint paragraph:

If the second and third operands are pointers into different address spaces, the address spaces must overlap.

**Section 6.5.15 - Conditional operator**, change the first sentence of paragraph 6 to:

If both the second and third operands are pointers or one is a null pointer constant and the other is a pointer, the result type is a pointer to a type qualified with all the access qualifiers of the referenced types of both operands, and qualified with the address-space qualifier for the combined address space of the referenced address spaces of the two operands or with no address-space qualifier if the combined address space is the generic one.

**Section 6.5.16.1 - Simple assignment**, change the third and fourth bullets of paragraph 1 to:

- both operands are pointers to qualified or unqualified versions of compatible types, the referenced address space of the left encloses the referenced address space of the right, and the referenced type of the left has all the access qualifiers of the referenced type of the right;
- one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **void**, the referenced address space of the left encloses the referenced address space of the right, and the referenced type of the left has all the access qualifiers of the referenced type of the right;

**Section 6.7.1 - Storage-class specifiers**, in paragraph 1, add to the syntax options for *storage-class-specifier*:

**register register-name**

**Section 6.7.1 - Storage-class specifiers**, add three new constraint paragraphs:

A named-register storage-class specifier shall only be used in the declaration of an object. The type of an object declared with a named-register storage-class specifier shall not be an array, shall not be qualified by an address-space qualifier, and shall be of a size that is not larger than the size of the named register.

If an object is declared with a named-register storage-class specifier, every declaration of that object shall include the same named-register storage-class specifier.

For any named register, there shall be in a translation unit no more than one object declared with the named-register storage-class specifier of that name.

**Section 6.7.1 - Storage-class specifiers**, add a new paragraph between existing paragraphs 4 and 5:

A storage-class specifier of the syntax **register** *register-name* is a *named-register storage-class specifier*, corresponding to the given register name. Register names are discussed in 6.7.1.1.

### **Section 6.7.1.1 - Named registers (NEW SECTION)**

An implementation may recognise some number of *named registers*, which are intended to be extraordinary storage locations that are not treated as ordinary address-space locations. Modifying the value of a named register may have additional, unspecified side effects. A named register has a implementation-defined fixed size and may or may not have an address in some address space.

Every named register has a unique name in the form of an identifier.

Some (possibly empty) implementation-defined set of named registers are *intrinsic registers*. The name of an intrinsic register shall begin with an underscore and an uppercase letter or with two underscores (and hence is a reserved identifier as defined in 7.1.3). There is no declaration for the name of an intrinsic register in a translation unit; the identifier is *implicitly declared* with a scope covering the entire translation unit.

An implementation may optionally support an implementation-defined syntax for declaring other (not intrinsic) named registers.

An object declared with a named-register storage-class specifier is allocated in the named register indicated by the specifier (thus, in effect, associating the object with the register). If the object has a size smaller than the register, the correspondence of bits between the object and the register is unspecified.

If more than one object in a program is declared with the same named-register storage-class specifier, the behavior is undefined.

**Section 6.7.2.1 - Structure and union specifiers**, add a new constraint paragraph:

The type of a member of a structure or union shall not be qualified with an address-space qualifier.

**Section 6.7.3 - Type qualifiers**, in paragraph 1, add to the syntax options for *type-qualifier*:

*address-space-name*

**Section 6.7.3 - Type qualifiers**, add three new constraint paragraphs:

No type shall be qualified by qualifiers for two or more different address spaces.

The type of an object with automatic storage duration shall not be qualified by an address-space qualifier.

An address-space qualified pointer shall not be a function pointer.

## **4 Basic I/O hardware addressing**

### **4.1 Rationale**

Embedded applications often must interact with specialised I/O devices, such as real-time sensors, motors, and LCD displays. At the lowest level, these devices are accessed and controlled through a set of special hardware registers (I/O registers) that device driver software can read and/or write.

Although different embedded systems typically have their own unique collections of hardware devices, it is not unusual for otherwise very different systems to have virtually identical interfaces to similar devices.

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 behaviour 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 the hardware itself can be connected.

#### **4.1.1 Basic Standardisation Objectives**

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

- A standardised interface must not prevent compilers from producing machine code that has no additional overhead compared to code produced by existing proprietary solutions. This requirement is essential in order to get widespread acceptance from the market place.
- The I/O driver source code modules should be completely portable to any processor system without any modifications to the driver source code being required [i.e. the syntax should promote I/O driver source code portability across different execution environments.]
- A standardised interface 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 standardisation 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.).]

### **4.2 Concepts**

To get an overview of the concepts related to basic I/O hardware addressing and to get a short definition of the terms related to basic I/O hardware addressing used in this Technical Report consider the following:

- **I/O** and **I/O** are short notations for Input-Output.
- An **I/O register** is the basic data unit in an **I/O device**.
- An **I/O device** is a hardware unit which uses **I/O registers** to create a data interface between a processor and the external world.
- An **I/O device driver** is software which operates on **I/O registers** in an **I/O device**.
- The **logical I/O register** is the register unit as it is seen from the I/O hardware. The C data type used for holding the I/O register data must have a *bit-width* equal to, or larger than, the *bit-width* of *logical I/O register*. The bit-width of the *logical I/O register* may be larger than the bit-width of the *I/O device* data bus or the processor data bus.
- **I/O register access** is the process of transferring data between an *I/O register* and one of the C compilers native data storage objects. This process is in a C program defined via an **I/O register designator** for the given *I/O register* or *I/O register buffer*.
- The **I/O register designator** is a unique identifier for the *I/O register*. It specifies **I/O access properties** related to the *I/O register* itself (for instance the *I/O register bit width* and *I/O register endian*) and properties related to the *I/O register access* method (for instance processor address space and address location).
- The main purpose with the *I/O register designator* is to **encapsulate I/O access properties** and hide this information from the *I/O device driver* code, in order to make the *I/O device driver* code independent of any particular processor (or compiler).
- Multiple *I/O registers* of equal size may form a **I/O register buffer**. All registers in the *I/O register buffer* is addressed using the same *I/O register designator*. An *I/O register buffer element* is referenced with an *index* similar to a C array.
- Multiple *I/O registers* which share common *I/O access properties* may form a **I/O register group**.
- An *I/O device* may contain multiple *I/O registers*. Such registers form an **I/O register group** which are portable as a single hardware unit (for instance an *I/O chip*, an *I/O cell*, a *plug-in board* etc).
- The common *I/O access properties* for the *I/O registers* in an *I/O register group* is defined by the **I/O group designator**.

Typical **I/O access properties** which are defined and encapsulated via the *I/O register designator*:

- The **access methods** used for *I/O register access*. *Access methods* refer to the various ways of addressing *I/O registers* and connecting *I/O devices* in a given processor architecture. Typical methods are *direct addressing*, *indexed addressing*, and addressing via **I/O access drivers**. Different methods have different *I/O access properties*. Common for all access methods is that all *access properties* are encapsulated by the *I/O register designator*.
- All *I/O registers* in the same **I/O register group** will use the same access method. Only the *I/O register characteristics* and address information will vary between the *I/O register designators*.
- If all the access properties defined by the *I/O register designator* can be initialised at compile time then the designator is called a **direct designator**.
- If some access properties defined by the *I/O register designator* is initialised at compile time and other require initialisation at run time then the designator is called an **indirect designator**. The *I/O driver* will need to know whether a designator is a *direct designator* or an *indirect designator* only for the purpose of mapping (initialising) an *I/O group designator*.

- If the bit-width of the *logical I/O register* is larger than the bit width of the *I/O device data bus*, then seen from the processor system the *logical I/O register* will consist of two or more **partial I/O registers**. In such cases the **I/O register endian** defines the MSB (Most Significant Byte), LSB (Least Significant Byte) order for the *partial I/O registers*. The *I/O register endian* is not related to any endian used by the processor system or compiler.
- If the bit-width of the *logical I/O register* is larger than the bit width of the processor data bus or the bit width of the *I/O device data bus*, then a single **logical I/O register access operation** will consist of multiple **partial I/O register access operations**. Such properties may be encapsulated by a single *I/O register designator* for the *logical I/O register*.

These concepts and terms are described in greater detail in the following sections.

## 4.3 Basic I/O-Hardware addressing header <iohw.h>

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

### 4.3.1 Standardisation principles

The I/O 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 of 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, device 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.

In the driver source code, an I/O register must always be referred with a symbolic name, the *ioregister\_designator*. The symbolic name must refer to a complete definition of the access method used with the given register. A standardised I/O syntax approach 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, only the definition of the access method (definition of the symbolic name) needs to be updated.

### 4.3.2 The abstract model

The standardisation of basic I/O hardware addressing is based on a three layer abstract model:

|                                               |
|-----------------------------------------------|
| The portable I/O device driver source code    |
| The users I/O register designator definitions |
| The vendors <iohw.h> implementation           |

The top layer contains the I/O *driver code* written by the compiler user. The source code in this layer is fully portable to any platform where the I/O hardware can be connected. This code must only access I/O hardware registers via the standardised I/O functions described in this section. Each I/O register must be identified using a symbolic name, the I/O *register designator*.

The middle layer contains the users specification of the I/O *register designators* used by the source code in the top layer. This layer associates a symbolic name with a complete access-specification for the I/O register in the given platform. The I/O *register designator* definitions in this layer is the only part which must be updated when the I/O *driver source code* is ported to a different platform.

The bottom layer is the implementation of the <iohw.h> header. It provides prototypes for the functions defined in this section and specifies the various different access methods supported by the given processor and platform architecture. This layer is typically implemented by the compiler vendor. The features provided by this layer, and used by the middle layer, may depend on intrinsic compiler functionality.

Annex C contains some general considerations, which should be addressed when a compiler vendor implements the iohw functionality.

Annex D proposes a generic syntax for I/O designator specifications. Using a general syntax on the middle layer may extend portability to include users I/O designator specification, so they can be used with different compiler implementations for the same platform.

#### 4.3.2.1 Structuring for I/O driver portability

The standardisation model above promotes that I/O driver portability is achieved by using minimum three modules, one for each of the abstract layers:

|                   |                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| I/O driver module | The I/O driver C source code. Portable across compilers and platforms. Includes the other header files below.                                                                                                                                                                                                                                                                                                   |
| "iodriv_hw.h"     | Specifies the I/O <i>register designators</i> used by the I/O driver module and maps the I/O <i>register designators</i> to an access method specific for the given execution environment.<br>The creator of the I/O driver module must only define the header file name and the symbolic names for the I/O <i>register designators</i> . The rest is implemented and maintained by the I/O driver module user. |
| <iohw.h>          | Standard header. Defines I/O functions in this standard and the access methods which can be used with I/O <i>register designator</i> specifications. Typically specific for a given compiler. Implemented by the compiler vendor.                                                                                                                                                                               |

Example:

```
#include <iohw.h>
#include "iodriv_hw.h" // 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.

#### 4.3.2.2 Typical information required by the user of a portable I/O driver module

In order to enable I/O driver library users later to define the I/O *register designators* and the I/O *group designators* for a specific platform, a portable I/O driver library based on the `<iohw.h>` interface should at least provide the following information to the library user, in addition to the library source code:

- All symbolic names for I/O registers and I/O groups used by the library.
- IO device and register type information for all symbolic names:
  - The bit width of the *logical I/O register*;
  - The designator type, is it the designator for a single I/O *register*, a I/O *register buffer* or a I/O *register group*;
  - Bit width of the I/O *device* data bus and the endian of I/O registers device (if the any register has a logical width larger than the I/O device data bus);
  - Relative address offset of registers in the device (if the device contains more than one register);
  - Whether the I/O device driver assumes the use of *indirect designators*.

## 4.4 Specifying I/O registers

For the purpose of this technical report, the C language is extended to include a new standard header `<iohw.h>` that provides support for accessing hardware I/O registers. Unlike the named registers of Section 3 of this technical report, it is not assumed an implementation has prior knowledge of all the I/O registers a program may access. Therefore, a program must in general supply information about an I/O register before the register can be accessed.

### 4.4.1 I/O-register designators

Within a C program, a machine's I/O registers are specified by *I/O-register designators*. An I/O-register designator may be an identifier or some other, implementation-specific construct. An implementation must support I/O-register designators in the form of identifiers; other forms of I/O-register designators may be supported but might not be portable to all implementations.

Any unique, non-reserved identifier can be defined as a designator for an I/O register. The definition of the identifier includes the size and access method of the I/O register. The means, however, by which an identifier is defined as an I/O-register designator is entirely implemented-defined.

By choosing convenient identifiers as designators for registers, a programmer can create device driver code with the expectation that the identifiers can be defined to refer to the actual I/O registers on a machine supporting the same interface. So long as the only important differences from one platform to another are the access methods for the registers, device driver code can be ported to a new platform simply by updating the designator definitions for the new platform.

Additional issues and recommendations concerning I/O-register designators are discussed in Annexes C, D, and E of this Technical Report.

#### 4.4.2 Accesses to individual I/O registers

The `<iohw.h>` header declares a number of functions and/or macros for accessing an I/O register given an I/O-register designator. Each “function” defined by the `<iohw.h>` header may actually be implemented either as a function or as a function-like macro that expands into an expression having the effects described for the function. If a function is implemented as a function-like macro, there will ordinarily not be a corresponding actual function declared or defined within the library.

The functions `iord` and `iordl` take an I/O-register designator argument and return a value read from the designated register. The register is read as an unsigned integer of the size of the register. This integer is then converted to the result type of the function (`unsigned int` for `iord` or `unsigned long` for `iordl`), and the converted value is returned as the function result.

For writing to registers, the functions `iowr` and `iowrl` take two arguments, an I/O-register designator and an unsigned integer, and write the integer to the designated register. The integer argument (`unsigned int` for `iowr` or `unsigned long` for `iowrl`) is converted to an unsigned integer of the size of the register, and this converted value is then written to the register. The result type of `iowr` and `iowrl` is `void`.

Example:

If `dev_status` and `dev_out` are I/O-register designators defined in the file `"iodriv_hw.h"`, the following is possible valid code:

```
#include <iohw.h>
#include "iodriv_hw.h" /* Platform-specific designator
```

```

    definitions. */

// Wait until controller is no longer busy.
while (iord(dev_status) & STATUS_BUSY) /* do nothing */;

// Write value to controller.
iowr(dev_out, ch);

```

Besides simple read and write operations, three read-modify-write operations are supported, corresponding to the bit-wise logical operations AND, OR, and XOR. The functions **ioand**, **ioor**, and **ioxor** take as arguments an I/O-register designator and an **unsigned int** value (the same arguments as **iowr**). For each function, the designated register is first read as if with **iord**. The corresponding bitwise logical operation is then performed between the value read and the integer argument, and this result is written back to the register as if with **iowr**. The result type of the functions is **void**. The functions **ioandl**, **ioorl**, and **ioxorl** are **unsigned long** versions of the same (a read is performed as if by **iordl**, and a write is performed as if by **iowl**).

#### 4.4.3 I/O register buffers

Besides individual I/O registers, an I/O-register designator may also designate an *I/O register buffer*, which is essentially an array of I/O registers. Like a C array, an integer index must be supplied to access a specific register in an I/O register buffer.

The **<iohw.h>** header declares the functions **iordbuf**, **iordbufl**, **iowrbuf**, **iowrbufl**, **ioorbuf**, **ioorbufl**, **ioandbuf**, **ioandbufl**, **ioxorbuf**, and **ioxorbufl**, corresponding to the functions for accessing individual I/O registers. These *-buf* versions of the access functions each take an additional argument supplying the index for the I/O register to access. The type of the index argument is **ioindex\_t**, an integer type defined in **<iohw.h>**.

Example:

If **ctrl\_buffer** is defined in the file **ctrl\_regs.h** as an I/O-register designator for an I/O register buffer, the following is possible valid code:

```

#include <iohw.h>
#include "ctrl_regs.h"      // Platform-specific designator
                           // definitions.

unsigned char buf[CTRL_BUFSIZE];

// Copy buffer contents.
for (int i = 0; i < CTRL_BUFSIZE; i++)
    buf[i] = iordbuf(ctrl_buffer, i);

```

Unlike an ordinary C array, the valid locations within an I/O register buffer might not be “dense”; any index might not correspond to an actual I/O register in the buffer. (A programmer should be able to determine the valid indices from documentation for the I/O interface or the machine.) If an I/O register buffer is accessed at an invalid index, the behaviour is undefined.

#### 4.4.4 I/O groups

An *I/O group* is an arbitrary collection of *I/O-register designators*. Each I/O group is intended to encompass all the designators for a single hardware device. Certain operations are supported only for I/O groups; these operations apply to the members of an I/O group as a whole. Whether an I/O-register designator can be a member of more than one group is implementation-defined.

Like I/O registers, an I/O group is specified by an *I/O-group designator*, which is an identifier or some other, implementation-specific construct. Any unique, nonreserved identifier can be defined as a designator for an I/O group. As with I/O-register designators, the means by which an identifier can be defined as an I/O-group designator is implemented-defined. Other forms of I/O-group designators may be supported but might not be portable to all implementations.

#### 4.4.5 Direct and indirect designators

Each I/O-register designator is either *direct* or *indirect*. An indirect I/O-register designator has a definition that does not fully specify the register or register buffer to which the designator refers. Before any accesses can be performed with it, an indirect designator must be *mapped* to refer to a specific register or register buffer. A direct I/O-register designator, by contrast, has a definition that fully specifies the register or register buffer to which the designator refers. A direct designator always refers to the same register or register buffer and cannot be changed.

An indirect I/O-register designator is mapped by associating it with a direct I/O-register designator. Accesses to the indirect designator then occur as though with the direct designator to which the indirect designator is mapped. An indirect I/O-register designator can be remapped any number of times; accesses through the designator always occur with respect to its latest mapping.

An implementation is not required to support indirect designators. If an implementation does support indirect designators, it may place arbitrary restrictions on the direct designators to which a specific indirect designator can be mapped. Typically, an indirect designator will be defined to be of a certain “kind”, capable of mapping to some subclass of access methods. An indirect designator can be mapped to a direct designator only if the direct designator’s access method is compatible with the indirect designator. Such issues are specific to an implementation.

For an I/O group, the members of the group must be either all direct designators or all indirect designators. An I/O-group designator is either direct or indirect, according to the members of the I/O group it designates.

#### 4.4.6 Operations on I/O groups

##### 4.4.6.1 Acquiring access to I/O registers

For some implementations, it may be necessary to *acquire* an I/O register or I/O register buffer before it can be accessed. What constitutes “*acquiring*” a register is specific to an implementation.

The `<iohw.h>` header declares two functions `iogroup_acquire` and `iogroup_release`, each taking a single direct I/O-group designator as an argument. The `iogroup_acquire` function acquires every register referred to by the group, and the `iogroup_release` function releases every register referred to by the group. If there is no need to acquire some I/O register or I/O register buffer, these functions have no effect for that register or register buffer.

Example

:

In an implementation for a hosted environment, an invocation of `iogroup_acquire` might call the operating system to map the physical I/O registers of the group into a block of addresses in the process's address space so that they can be accessed. In the same implementation, an invocation of `iogroup_release` would call the operating system to unmap the I/O registers, making them inaccessible to the process.

##### 4.4.6.2 Mapping indirect designators

The `<iohw.h>` header declares a function `iogroup_map` taking two arguments, the first an indirect I/O-group designator and the second a direct I/O-group designator. The function maps each indirect I/O-register designator in the first group to a corresponding direct I/O-register designator in the second group. The correspondence between members of the two I/O groups is determined in an implementation-defined way. (For example, if I/O-group designators are defined with their members listed in a particular order, the correspondence could be determined by matching the first member of one group with the first member of the other, etc.)

Example:

If `dev_hw.h` defines two indirect I/O-register designators, `dev_config` and `dev_data`, an indirect I/O-group designator `dev_group` with both `dev_config` and `dev_data` as members, and two direct I/O-group designators `dev1_group` and `dev2_group`, the following is possible valid code:

```
#include <iohw.h>
#include "dev_hw.h"    // Platform-specific designator
                      // definitions.

// Portable device driver function.
uint8_t get_dev_data(void)
{
    iowr(dev_config, 0x33);
    return iord(dev_data);
```

```

}

// Read data from device 1.
iogroup_map(dev_group, dev1_group);
uint8_t d1 = get_dev_data();

// Read data from device 2.
iogroup_map(dev_group, dev2_group);
uint8_t d2 = get_dev_data();

```

#### 4.5 Detailed changes to ISO/IEC 9899:1999

This section details the modifications to ISO/IEC 9899:1999 needed to incorporate the basic I/O hardware addressing functionality overviewed in Sections 4.1 through 4.4 of this Technical Report. The changes listed in this section are limited to syntax and semantics; examples, forward references, and other descriptive information are omitted.

One new section is added to ISO/IEC 9899:1999 covering the new library header `<iohw.h>`. No changes to other sections are needed.

#### Section 7.8a - Basic I/O hardware addressing "`<iohw.h>`" (NEW SECTION)

The header `<iohw.h>` declares a type and defines macros and/or declares functions for accessing implementation-specific I/O registers.

The type declared is

`ioindex_t`

which is the unsigned integer type of an index into an I/O register buffer.

Any “function” declared in `<iohw.h>` as described below may alternatively be implemented as a function-like macro defined in `<iohw.h>`. (If a function in `<iohw.h>` is implemented as a function-like macro, there need not be an actual function declared or defined as described, despite the use of the word *function*.) Any invocation of such a function-like macro shall expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments.

##### 7.8a.1 - I/O registers

An *I/O register* is a storage location that is addressable within some address space. An I/O register has a size and an *access method*, which is the method by which an implementation accesses the register at execution time. An I/O register is accessed (read or written) as an unsigned integer. An I/O register may need to be *acquired* before it can be accessed. (I/O registers are acquired with the `iogroup_acquire` function described in 7.8a.3.1.)

Accesses to an I/O register may have unspecified side effects that may be unknown to the implementation, and an I/O register may be modified in ways unknown to the implementation. Accesses to I/O registers performed by functions declared in `<iohw.h>` are therefore treated as side effects which respect sequence points.

An *I/O register buffer* is a collection of I/O registers indexed by an integer of type `ioindex_t` and otherwise sharing a common size and access method. The set of valid indices for the I/O registers in an I/O register buffer may be any subset of the nonnegative integers of type `ioindex_t`; the set of valid indices need not be contiguous and need not include zero.

An *I/O-register designator* refers (except as stipulated below) to a specific individual I/O register or a specific I/O register buffer. Functions that access I/O registers take an I/O-register designator argument to determine the register to access. An implementation shall support at least one of the following as a valid I/O-register designator for any individual I/O register or I/O register buffer:

- any ordinary identifier that is not a reserved identifier, defined by some implementation-defined means; and/or
- any object-like macro name that is not a reserved identifier, defined in accordance with some implementation-defined convention.

An implementation may optionally support other, implementation-defined forms of I/O-register designators.

An implementation may limit an identifier that is an I/O-register designator to some number of characters, this limit must not be less than 25 characters. (Each universal character name or extended source character is considered a single character.)

Each I/O-register designator is either *direct* or *indirect*. A *direct* I/O-register designator refers to a specific I/O register or I/O register buffer as determined by the designator's definition. An *indirect* I/O-register designator does not refer to a specific I/O register or I/O register buffer until the designator has been *mapped* to a direct I/O-register designator. Once mapped, an indirect I/O-register designator can subsequently be remapped (mapped again) to the same or a different direct I/O-register designator. An indirect I/O-register designator refers to the same I/O register or I/O register buffer as the direct designator to which it was last mapped. (I/O-register designators are mapped with the `iogroup_map` function described in 7.8a.3.2.)

An indirect I/O-register designator is *compatible with* a direct I/O-register designator if it is possible to map the indirect designator to the direct designator. An I/O-register designator that refers to an individual I/O register is not compatible with an I/O-register designator that refers to an I/O register buffer, and vice versa. Otherwise, whether a specific indirect I/O-register designator is compatible with a specific direct I/O-register designator is implementation-defined.

An implementation need not support a means for indirect I/O-register designators to be defined.

An I/O-register designator *covers* an I/O register if it refers to the I/O register or it refers to an I/O register buffer that includes the register.

### 7.8a.2 - I/O groups

An *I/O group* is a collection of I/O-register designators. It is intended that each I/O group encompass all the designators for a single hardware controller or device.

The members of an I/O group shall be either all direct designators or all indirect designators. An I/O group is *direct* if its members are direct. An I/O group is *indirect* if its members are indirect.

An I/O group shall not have as members two or more I/O-register designators that cover the same I/O register. Whether an I/O-register designator can be a member of more than one I/O group at the same time is implementation-defined.

An *I/O-group designator* specifies an I/O group. An implementation shall support at least one of the following as a valid I/O-group designator for any supported I/O group:

- any ordinary identifier that is not a reserved identifier, defined by some implementation-defined means; and/or
- any object-like macro name that is not a reserved identifier, defined in accordance with some implementation-defined convention.

[An implementation may optionally support other, implementation-defined forms of I/O-group designators. An implementation may limit an identifier that is an I/O-group designator to some number of characters, this limit must not to be less than 25 characters. (Each universal character name or extended source character is considered a single character.)]

### 7.8a.3 - I/O group functions

#### 7.8a.3.1 - The `iogroup_acquire` and `iogroup_release` functions

##### Synopsis

```
#include <iohw.h>
void iogroup_acquire( _iogroup_designator_ )
void iogroup_release( _iogroup_designator_ )
```

##### Description

The `iogroup_acquire` function acquires a collection of I/O registers; the `iogroup_release` function releases a collection of I/O registers. *Releasing* an I/O register undoes the act of acquiring the register. The functions acquire or release all the I/O registers covered by the I/O-register designators that are members of the I/O group designated by `_iogroup_designator_`. If the I/O group is indirect, the behaviour is undefined.

An I/O register is only said to be *acquired* between an invocation of **iogroup\_acquire** that acquires the register and the next subsequent invocation of **iogroup\_release**, if any, that releases the register. If **iogroup\_release** releases an I/O register that is not at the time acquired, or if **iogroup\_acquire** acquires an I/O register that is at the time already acquired, the behaviour is undefined.

Acquiring or releasing an I/O register is treated as a side effect which respects sequence points.

If an implementation can access a particular I/O register without needing it to be first acquired, the act of *acquiring* and the act of *releasing* the register may have no real effect.

### 7.8a.3.2 - The **iogroup\_map** function

#### Synopsis

```
#include <iohw.h>
void iogroup_map( _iogroup_designator_, _iogroup_designator_ )
```

#### Description

The **iogroup\_map** function maps the indirect I/O-register designators in the I/O group designated by the first **\_iogroup\_designator\_** to corresponding direct I/O-register designators in the I/O group designated by the second **\_iogroup\_designator\_**. The first I/O group shall be indirect, and the second I/O group shall be direct. The correspondence between members of the two I/O groups is implementation-defined and shall be one-to-one. If an indirect I/O-register designator is mapped to a direct I/O-register designator with which it is not compatible, the behaviour is undefined.

### 7.8a.4 - I/O register access functions

If a register is accessed (read or written) when it is not acquired, the behaviour is undefined. If an indirect I/O-register designator is given as an argument to one of the functions below and the designator has not been mapped, the behaviour is undefined.

#### 7.8a.4.1 - The **iord** functions

#### Synopsis

```
#include <iohw.h>
unsigned int iord( _ioreg_designator_ )
unsigned long iordl( _ioreg_designator_ )
```

#### Description

The functions **iord** and **iordl** read the individual I/O register referred to by **\_ioreg\_designator\_** and return the value read. The I/O register is read as an unsigned

integer of its size; the read value is then converted to the result type, and this converted value is returned.

#### 7.8a.4.2 - The iordbuf functions

##### Synopsis

```
#include <iohw.h>
unsigned int iordbuf( _ioreg_designator_, ioindex_t ix )
unsigned long iordbufl( _ioreg_designator_, ioindex_t ix )
```

##### Description

The functions **iordbuf** and **iordbufl** read one of the I/O registers in the I/O register buffer referred to by **\_ioreg\_designator\_** and return the value read. The functions are equivalent to **iord** and **iordl**, respectively, except that the I/O register read is the one with index **ix** in the I/O register buffer referred to by **\_ioreg\_designator\_**. If **ix** is not a valid index for the I/O register buffer, the behaviour is undefined.

#### 7.8a.4.3 - The iowr functions

##### Synopsis

```
#include <iohw.h>
void iowr( _ioreg_designator_, unsigned int a )
void iowrl( _ioreg_designator_, unsigned long a )
```

##### Description

The functions **iowr** and **iowrl** write the individual I/O register referred to by **\_ioreg\_designator\_**. The unsigned integer **a** is converted to an unsigned integer of the size of the I/O register, and this converted value is written to the I/O register.

#### 7.8a.4.4 - The iowrbuf functions

##### Synopsis

```
#include <iohw.h>
void iowrbuf( _ioreg_designator_, ioindex_t ix, unsigned int a )
void iowrbufl( _ioreg_designator_, ioindex_t ix, unsigned long a )
```

##### Description

The functions **iowrbuf** and **iowrbufl** write one of the I/O registers in the I/O register buffer referred to by **\_ioreg\_designator\_**. The functions are equivalent to **iowr** and **iowrl**, respectively, except that the I/O register written is the one with index **ix** in the I/O register buffer

referred to by `_ioreg_designator_`. If `ix` is not a valid index for the I/O register buffer, the behaviour is undefined.

#### 7.8a.4.5 - The `ioor`, `ioand`, and `ioxor` functions

##### Synopsis

```
#include <iohw.h>
void ioand( _ioreg_designator_, unsigned int a )
void ioor( _ioreg_designator_, unsigned int a )
void ioxor( _ioreg_designator_, unsigned int a )

void ioorl( _ioreg_designator_, unsigned long a )
void ioandl( _ioreg_designator_, unsigned long a )
void ioxorl( _ioreg_designator_, unsigned long a )
```

##### Description

The functions `ioand`, `ioandl`, `ioor`, `ioorl`, `ioxor`, and `ioxorl` modify the individual I/O register referred to by `_ioreg_designator_`. The function `ioand` has a behaviour equivalent to

```
iowr( _ioreg_designator_, iord( _ioreg_designator_ ) & a )
```

except that the `_ioreg_designator_` is not evaluated twice (assuming it is an expression). Likewise, the function `ioor` has a behaviour equivalent to

```
iowr( _ioreg_designator_, iord( _ioreg_designator_ ) | a )
```

and the function `ioxor` has a behaviour equivalent to

```
iowr( _ioreg_designator_, iord( _ioreg_designator_ ) ^ a )
```

Corresponding equivalencies apply for `ioandl`, `ioorl`, and `ioxorl`, but with the `unsigned long` functions `iordl` and `iowrl` replacing `iord` and `iowr`.

#### 7.8a.4.6 - The `ioorbuf`, `ioandbuf`, and `ioxorbuf` functions

##### Synopsis

```
#include <iohw.h>
void ioandbuf( _ioreg_designator_, ioindex_t ix, unsigned int a )
void ioorbuf( _ioreg_designator_, ioindex_t ix, unsigned int a )
void ioxorbuf( _ioreg_designator_, ioindex_t ix, unsigned int a )

void ioandbufl( _ioreg_designator_, ioindex_t ix, unsigned long a )
```

```
void ioorbufl( _ioreg_designator_, ioindex_t ix, unsigned long a )
void ioxorbufl( _ioreg_designator_, ioindex_t ix, unsigned long a )
```

### Description

The functions **ioandbuf**, **ioorbuf**, **ioxorbuf**, **ioorbufl**, **ioandbufl**, and **ioxorbufl** modify one of the I/O registers in the I/O register buffer referred to by **\_ioreg\_designator\_**. The functions are equivalent to **ioand**, **ioandl**, **ioor**, **ioorl**, **ioxor**, and **ioxorl**, respectively, except that the I/O register modified is the one with index **ix** in the I/O register buffer referred to by **\_ioreg\_designator\_**. If **ix** is not a valid index for the I/O register buffer, the behaviour is undefined.

## Annex A

### Additional information and Rationale

#### A.1 Fixed-point

##### A.1.1 The fixed-point data types

The set of representable floating-point values (which is a subset of the real values) is characterised by a sign, a precision and the position of the radix point. For those values that are commonly denoted as floating-point values, the characterising 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 generalisation to other radices (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 characterised 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 data types (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 characterised 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 data value (and hence the value of that fixed-point data value!) by simply looking at the representation of that data value. 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 data type) to an infinite number of different fixed-point data types (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 data types, and, unfortunately, each hardware manufacturer has made its own selection, depending on the field of application of the processor implementing the fixed-point data type. Unless a complete dynamic or a parameterised typesystem is used (not part of the current C standard, hence not proposed here), for programming language support of fixed-point data types a number of choices need to be made to limit the number of allowable (and/or supported or to be supported) fixed-point data types. In order to give some guidance for those choices, some aspects of fixed-point data values and their uses are investigated here.

For the sake of this discussion, a fixed-point data value 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 data type with a larger (positive) maximum value.

Note that the size of (the number of bits used for) a fixed-point data value does not necessarily equal the size of the container in which the fixed-point data value is contained (or through which the fixed-point data value is addressed): there may be gaps here!

### A.1.2 Classification of Fixed Point Types

As stated before, it is necessary, when using a fixed-point data value, 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 data type, and does not (for this discussion) form part of the fixed-point data types.

- 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 data type (for large, but not very precise integer values) that is normally not supported by (fixed-point) hardware. In this document, these fixed-point data types 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 data type with more bits; in any case this data type can be simulated by using appropriate normalise (shift left/right) operations. There is no known fixed-point hardware that supports this data type.
- The radix point is located immediately to the left of the leftmost (most significant) bit of the databits. This data type has values (for signed data types) in the interval (-1,+1), or (for unsigned data types) in the interval [0,1]. This is a very common, hardware supported, fixed-point data type. In the rest of this section, this fixed-point data type will be called the type-A fixed-point data type. Note that for each number of databits, there are one (signed) or two (signed and unsigned) possible type-A fixed-point data types.
- The radix point is located somewhere between the leftmost and the rightmost bit of the databits. The data values for this fixed-point data type have an integral part and a fractional part. Some of these fixed-point data types are regularly supported by hardware. In the rest of this section, this fixed-point data type will be called the type-B fixed-point data type. For each number of databits N, there are (N-1) (signed) or (2\*N-1) (signed and unsigned) possible type-B fixed-point data types.

Apart from the position of the radix point, there are three more aspects that influence the amount of possible fixed-point data types: the presence of a signbit, the number of databits comprising the fixed-point data values and the size of the container in which the fixed-point data values are stored. In the embedded processor world, support for unsigned fixed-point data types is rare; normally only signed fixed-point data types are supported. However, to disallow unsigned 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 data types:

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

### A.1.3 Recommendations for Fixed Point Types in C

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

1. Introduce **signed** and **unsigned** fixed-point data types, and use the existing signed and unsigned keywords (in the 'normal' C-fashion) to distinguish these types. Omission of either keyword implies a signed fixed-point data type.
2. Introduce a new keyword and *type-specifier \_Fract* (similar to the existing keyword **int**), and define the following three standard signed fixed-point types (corresponding to the type-A fixed-point data types, as described above): **short \_Fract**, **\_Fract** and **long \_Fract**. The supported (or required) underlying fixed-point data types 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 data value. Note that there is not necessarily a correspondence between a fixed-point data type designator and the type of its container: when an 18-bit and a 30-bit fixed-point data type are supported, the 18-bit will probably have the **short \_Fract** type and the 30-bit type will probably have the **\_Fract** type, while the containers of these types will be the same.
3. Introduce a new keyword and *type-specifier \_Accum*, and define the following three standard signed fixed-point types (corresponding to the type-B fixed-point data types, as described above): **short \_Accum**, **\_Accum** and **long \_Accum**, with similar representation requirements as for the **\_Fract** type.
4. If more fixed-point data types are needed, (or if there is a need to better distinguish certain fixed-point data types), an approach similar to the **<stdint.h>** approach could be taken, whereby **fract\_leN\_t** could designate a (type-A) fixed-point data type with at least N databits, while **fract\_leM\_leN\_t** could designate a (type-B) fixed-point data type with at least M integral bits and N fractional bits. Note that the introduction of these generalised fixed-point data types is currently not included in the main text of this Technical Report.
5. In order for the programmer to be able to write portable algorithms using fixed-point data types, information on (and/or control over) the nature and precision of the underlying fixed-point data types should be provided. The normal C-way of doing this is by defining macro names (like **SFRACT\_FBIT** etc.) that should be defined in an implementation-defined manner.

The C standard , with its defined keywords, allows for yet another size for fixed-point data types: **long long fract**. The specified three sizes were considered to be enough for the current systems, the **long long** variant might, for the time being, be added by an implementation in an implementation-defined manner.

#### A.1.4 Number of Fractional data bits in `_Fract` versus `_Accum`

At some point it was considered to require that the number of fractional bits in a `_Fract` type would be exactly the same as the number of fractional bits in the corresponding `_Accum` type. The reason for this was that `_Accum` can be viewed upon as the placeholder sums of `_Fracts`. This requirement would be fulfilled for implementations on typical DSP processors. However, it was chosen not to make this a strict requirement because other machine classes would have trouble using their hardware supported data types when implementing the fractional data types. A discussion on this issue is given below.

#### **A type for accumulating sums cannot always be fixed at the same number of fractional bits as the associated fractional type.**

*Many SIMD architectures do not support fixed-point types that have the same number of fractional bits as a fractional type, plus some integer bits. To manufacture accumulator types that are not supported by the hardware would add overhead and often require a loss of parallelism. Also, often there is no way to detect a carry out of a packed data type, so even the simple implementation of providing one SIMD word of fractions plus one SIMD word of integer bits is not always available.*

*In addition, manufacturing accumulator types of artificial widths is usually unnecessary since there are already accumulator types supported by the hardware. This means that the language needs to be flexible enough to allow the existing hardware-supported data types to be used rather than imposing a strict model that hampers performance.*

*For example, Radiax pairs 16-bit objects into a 32-bit SIMD word. The accumulator type provided for arithmetic on these objects is 40 bits wide per object, composed of 32 fractional bits and 8 integer bits. There is no other accumulator type supported. An artificial requirement that exactly 16 fractional bits be available would severely impact performance, and would have the surprising effect that addition would become much slower than multiplication.*

*In the VIS architecture, the supported hardware types that can be used as accumulation types sometimes have more fractional bits than the underlying fractional types, and sometimes fewer, but never the same number. Also, there is no direct path between SIMD registers (which overload the floating-point registers) and the integer registers, so constructing an artificial type involves not only a loss of parallelism but also extra loads and stores to move data between the SIMD registers and the integer registers.*

*The proposal to fix an accum's fractional bits at the same number as the underlying fract type is therefore prohibitively expensive on some architectures and needs to be removed.*

#### A.1.5 Possible Data Type Implementations

By way of example, these tables show the fixed-point formats we would suggest for various classes of processors:

|                           | --- signed _Fract ---   |        |      | --- signed _Accum ---   |        |        |
|---------------------------|-------------------------|--------|------|-------------------------|--------|--------|
|                           | short                   | middle | long | short                   | middle | long   |
| typical desktop processor | s.7                     | s.15   | s.31 | s8.7                    | s16.15 | s32.31 |
| typical 16-bit DSP        | s.15                    | s.15   | s.31 | s8.15                   | s8.15  | s8.31  |
| typical 24-bit DSP        | s.23                    | s.23   | s.47 | s8.23                   | s8.23  | s8.47  |
| Intel MMX                 | s.7                     | s.15   | s.31 | s8.7                    | s16.15 | s32.31 |
| PowerPC AltiVec           | s.7                     | s.15   | s.31 | s8.7                    | s16.15 | s32.31 |
| Sun VIS                   | s.7                     | s.15   | s.31 | s8.7                    | s16.15 | s32.31 |
| MIPS MDMX                 | s.7                     | s.15   | s.31 | s8.7                    | s8.15  | s17.30 |
| Lexra Radiax              | s.7                     | s.15   | s.31 | s8.7                    | s8.15  | s8.31  |
| ARM Piccolo               | s.7                     | s.15   | s.31 | s8.7                    | s16.15 | s16.31 |
|                           |                         |        |      |                         |        |        |
|                           | --- unsigned _Fract --- |        |      | --- unsigned _Accum --- |        |        |
|                           | short                   | middle | long | short                   | middle | long   |
| typical desktop processor | .8                      | .16    | .32  | 8.8                     | 16.16  | 32.32  |
| typical 16-bit DSP        | .16                     | .16    | .32  | 8.16                    | 8.16   | 8.32   |
| typical 24-bit DSP        | .24                     | .24    | .48  | 8.24                    | 8.24   | 8.48   |
| Intel MMX                 | .8                      | .16    | .32  | 8.8                     | 16.16  | 32.32  |
| PowerPC AltiVec           | .8                      | .16    | .32  | 8.8                     | 16.16  | 32.32  |
| Sun VIS                   | .8                      | .16    | .32  | 8.8                     | 16.16  | 32.32  |
| MIPS MDMX                 | .8                      | .16    | .32  | 8.8                     | 8.16   | 16.32  |
| Lexra Radiax              | .8                      | .16    | .32  | 8.8                     | 8.16   | 8.32   |
| ARM Piccolo               | .8                      | .16    | .32  | 8.8                     | 16.16  | 16.32  |

(The "typical" DSPs referred to in the table cannot address units in memory smaller than 16 or 24 bits, which is why these processors aren't expected to support a **short \_Fract** smaller than **\_Fract**.)

### A.1.6 Overflow and Rounding

Fixed-point data types are often used in situations where floating-point data types otherwise would have been used. Typically because the underlying hardware does not support floating-point operations directly for various reasons. One important property of fixed-point data types is the fixed dynamic range. To exploit the dynamic range, data are often scaled so overflow will occur with a certain likelihood. Therefore overflow behaviour is important for fixed-point data types. Saturation on overflow is often preferred. Furthermore saturation on overflow is often supported in hardware by processors that naturally operate on fixed-point data types. Therefore, a type qualifier **\_Sat** to specify saturation on overflow for the new fixed-point types is proposed in this TR.

It was decided to give the programmer control over the general behaviour of operands declared without an explicit fixed-point overflow type-qualifier by two pragmas, **FX\_FRACT\_OVERFLOW** and **FX\_ACCUM\_OVERFLOW**, for **fract** and **accum** types respectively. The default behaviour of this pragma is default (implying an undefined behaviour), but a programmer can choose to change the default behaviour to saturation on overflow with these pragmas. This is subject to the usual scoping rules for pragmas.

Generally it is required that if a value cannot be represented exactly by the fixed-point type, it should be rounded up or down to the nearest representable value in either direction. It was chosen not to specify this further as there is no common path chosen for this in hardware implementations, so it was decided to leave this implementation defined.

The above requirement to precision means that the result should be within 1 unit in last place (ulp). Processors that support fixed-point arithmetic in hardware have no problems in meeting this precision requirement without loss of speed. However, processors that will implement this with integer arithmetic may suffer a speed penalty to get to the 1 ulp result.

One such type of processors would be mainstream 32-bit processors, on which "long fract" might reasonably be implemented as 32 bits (format s.31). A multiplication of two 32-bit "long fract"s to a "long fract" result would typically be compiled as a  $32 \times 32 \rightarrow 64$ -bit integer multiplication followed by a shift right by 31 bits, keeping only the bottom 32 bits at the end. On many of these processors, the 64-bit product would be obtained in two 32-bit registers---say, R0 and R1---and then the 31-bit shift across the register pair would take three instructions:

```
shift R0 left 1 bit
shift R1 right 31 bits (an unsigned shift)
OR R1 into R0
```

which leaves the 32-bit "long fract" result in R0. But note that the most significant 31 bits of the result are already available in R0 after the first shift; the other two instructions serve only to move the last, least significant bit into position. If the product is permitted to be up to 2 ulps in error, an implementation could choose instead to leave the least significant bit zero and dispense with the last two instructions.

Although a 1-ulp bound is preferable the above example shows significant savings that will justify a larger bound. Therefore the user is allowed to choose speed over precision with a pragma, **FX\_FULL\_PRECISION**. The default state of this pragma is implementation-defined.

### A.1.7 Type conversions, usual arithmetic conversions

The fixed-point data types are positioned 'between' the integer data types and the floating-point data types: if only integer data types 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 data types, otherwise everything will be converted to the appropriate floating-point data type.

Since it is likely that an implementation will support more than one (type-A and/or type-B) fixed-point data type, in order to assure arithmetic consistency it should be well-defined to which fixed-point

data type a type is converted to before an operation involving fixed-point and integer data values is performed. There are several approaches that could be followed here:

- define that the result of any operation on fixed-point data types 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 an extended fixed-point data type 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 data type (if supported), or the 'largest' type-A fixed-point data type.

For the combination of an integer type and a fixed-point type, or the combination of a **\_Fract** type and an **\_Accum** type, the usual arithmetic conversions may lead to useless results or to gratuitous loss of precision. Consider the case of converting an integer to a **\_Fract** type. This will only be useful for the integer values 0 and -1. The same case can be made for mixing **\_Fract** and **\_Accum** types. Therefore the approach taken was to define that the result of any operation involving fixed-point types should be as if the operation is done using infinite precision. This deviates from the usual arithmetic conversions, in that no common type whereto both operands are converted is defined. Rather it can be said that the operation is performed directly with the value of the two operands, without any change in value due to usual arithmetic conversions. The resulting value of the operation is then subject to overflow and rounding as specified by its result type.

The mentioned approach gives the expected results for multiplication and division operations involving an integer and a fixed-point type operand, and is also used for addition and subtraction operations. In these latter cases, when the fixed-point type is a **\_Fract** type, the operation will result in an overflow for practically all values of the integer operand. It was decided not to disallow the combinations of integer and fixed-point type operands for addition and subtraction operations, but to encourage implementations to produce a warning when these operations are encountered.

## A.1.8 Operations involving fixed-point types

The decision not to promote integers to fixed-point to balance the operands is clearly a departure from the way C is normally defined and, in particular, the way the same operations work when integer and floating-point operands are mixed. The inconsistency has been introduced because integer values often cannot be promoted honestly to fixed-point types. None of the **\_Fract** types has *any* integer bits, and an implementation may have as few as four integer bits in its **\_Accum** types.

On such an implementation, it is impossible to convert an integer with a value larger than 8 to any fixed-point type, which leaves only a limited range of integers to work with. Consider, for example, the problem of dividing a fixed-point value by a (non-constant) integer value which could be as large as 15.

The floating-point types have the property that (on all known machines) the *range* of all the integers fits within even the smallest floating-point type, so converting an integer to floating-point at worst suffers a rounding error (and often not even that). This is definitely not the case for the fixed-point

types. On the other hand, unlike with floating-point, fixed-point and integer values have very similar representations, and their operations have similar implementations in hardware. Thus, it is less trouble for an implementation to mix integer and fixed-point operands and perform the calculation directly than it would be for floating-point.

The result type of an operation involving fixed-point types is the type with the higher rank. When mixing integer and fixed-point types, fixed-point types are chosen to have higher rank. The reason for this choice is based on the common case where a fixed-point value is multiplied by a factor of 2, and when a fixed-point type is divided by an integer value. The natural result type in this case is the fixed-point type.

As specified in the section "Overflow and Rounding", two types of overflow handling are defined for the fixed-point types: `_Sat` and default. Generally, if either operand has the `_Sat` qualifier, the result type of the operation will have the `_Sat` qualifier.

#### A.1.9 Exception for 1 and -1 Multiplication Results

The rule about 1 and -1 multiplication results is needed to permit an important optimisation for sum-of-products calculations on many DSPs (sum-of-products being primarily what DPSs are designed to do). Using the `long accum` type for the accumulator that holds the running sum, a sum-of-products (or dot product) can be naturally coded as:

```
fract a[N], b[N];
long accum acc = 0;
for ( ix = 0; ix < N; ++ix ) {
    acc += (long accum) a[ix] * b[ix];
}
```

While the above would be the obvious code, on many DSPs the multiply-accumulate hardware really does this:

```
acc += (long accum) ( (sat long fract) a[ix] * b[ix] );
```

In other words, the product is saturated to the `long fract` format before being added into the accumulator. The only detectable difference between this and the code above occurs when "a[ix]" and "b[ix]" are both -1, in which case the product is 1, which cannot be represented exactly as a `long fract`. In this case (and only this case), the DSP hardware saturates the 1 to the maximum `long fract` value before adding.

With the original code above, the rules in the section on "Overflow and Rounding" require that the product be represented exactly if the result type permits it. Since a 1 can always be represented exactly by a `long accum`, the rounding rules do not permit the 1 to be replaced by the maximum `long fract` value. (Note that the `long fract` type makes no appearance in the original code.) Unfortunately, on processors that only support sum-of-product operations that saturate the product to `long fract`, it is not possible to implement the code above efficiently as written.

without some compromise. Rather than relax the rounding rules in general, a special case has been made to cover this condition.

#### **A.1.10 Linguistic Variables and `unsigned _Fract`: an example of `unsigned` fixed-point**

The Linguistic variables definition and manipulation have been identified as a major application area that use `unsigned _Fract` variables as a storage requirement. Linguistic variables normalise the data of interest to an application between the values of 0 and 1. In applications the value associated with a linguistic variable is an expression of a degree of truth unlike boolean variables that take on the value of either 0 or 1. Fuzzy logic functions combining linguistic variables also return logic values scaled between 0 and 1.

Linguistic variables are used in many non-linear complex applications, in consumer applications, non-linear process control, animation, pattern recognition and financial applications.

Applications that use linguistic variables depend on the size of a linguistic variable in the application to determine the precision in which the results are computed. Applications that use linguistic variables select a resolution to be used throughout the application. Typical linguistic variables in commercial applications are 8 or 16 bits in size. Other sizes have been seen in commercial applications but would be impacted less by using their signed counterpart.

## Annex B

### **Embedded systems extended memory support.**

#### **B.1 Embedded systems extended memory support**

##### **B.1.1 Modifiers for named address spaces**

Applications on small-scale embedded systems run in a non-hosted environment, and on resource-constrained systems. Compilers for such systems are responsible for freeing the application developer from most, but not all, target-specific responsibilities. Embedded systems, including most consumer electronics products and DSP-driven devices, are optimised to support the requirements of their intended applications. Their central processors generally contain many separate address spaces. C language support for these systems extends the C linear address space to an address space that, although linear within memory spaces, is not always created equal. Application developers need the vocabulary to efficiently express how their application uses the target hardware.

Named address space type modifiers allow the application developer to express a very specific requirement, that variables be associated with a specific memory space. In turn, the compiler will be able to generate more efficient code for the target implementation.

##### **B.1.1.1 User-defined device drivers**

Many embedded systems include memory that can only be accessed with some form of device driver. These include memories accessed by serial data busses ( $I^2C$ , SPI), and on-board non-volatile memory that must be programmed under software control. 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 needs 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 functions take 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.
  - All memory in a specific named address space may not necessarily have contiguous addressing. It is common to find that memory, even though accessed through a

common means, may have gaps in its structure. In similar way some named address space memory may have physical address aliases.

- 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.
- 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 and structures.
  - Users need to direct the compiler to use a specific memory area.
  - The compiler needs to be free to use a user-defined memory area as a generic, general-purpose memory area, for the purpose of a variable spill area.

## B.1.2 Application-defined multiple address space support

Inherent processor-architecture-based address spaces are implementation-defined and provided by the compiler for that processor architecture and as such will be available to all applications normally supported by the processor. Examples include accesses to various native data spaces, or data spaces where write and read operations are not symmetrical (f.i., flash memories where read operations may run at full speed, while write operations occur through some driver code).

User-defined named address spaces are part of an application specific address space. The support code for user defined address space may very well be portable across many applications and quite possibly many different processors, but its nature is essentially part of an application . Data access to user defined named address space are often through I2C, microwire, or compact flash memory parts. Accesses to 'normal' address spaces may be handled by the compiler or may be resolved by the linker, for the user defined address spaces the modifier names need to be associated with user supplied access code.

The pragma **addressmod** is a method to encapsulate the memory declaration, to tie variable declarations to device drivers, and to provide the compiler the information necessary to generate the code that is required to access the variables that are declared by an application. User-defined memory could be global in nature, or local to one program segment.

Typical implementation of the address space modifiers are in project application specific headers files.

```
#pragma addressmod memory_space_name,
    Read_access,
    Write_access,
    [ optional additional address base and ranges ]
;
```

Read and write access is assumed to be byte wide. The resolution of data types other than character size is implementation-defined and may be resolved either in the compiler or the read and write device drivers.

**Read\_access** points to a user defined macro, function or inline function that the compiler will use to read variables assigned by the compiler in the named memory space.

**Write\_access** points to user defined macro, function or inline function that is used to read from the defined memory space.

Address base and size information is optional and implementation-defined. In many systems it may be defined at link time as part of the conventional linking process or at compile time.

Note that this is perhaps an odd place to define physical address space. Named address space can be application specific address space or simply a name designed to group variables for some common purpose. The generated code especially if either an inline function or macro is used for an access definition may be significantly optimised in the absence of a good optimising linker through a compile time optimisation familiar with specific address information. It is possible to define physical address space information at the linking phase in the traditional manner.

## Annex C

### Implementing the `<iohw.h>` header

#### C.1 General

The `<iohw.h>` header defines a standardised function syntax for basic I/O hardware addressing. This header should normally be created by the compiler vendor.

While this standardised 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, the `<iohw.h>` header itself may require careful consideration to achieve an efficient implementation.

This section gives some guidelines for implementers on how to implement the `<iohw.h>` header in a relatively straightforward manner given a specific processor and bus architecture.

Other approaches for implementing the `<iohw.h>` header are under consideration; in a later version of this document, such other approaches may be included next to, or as replacement of, the approach given here.

#### C.1.1 Recommended steps

Briefly, the recommended steps for implementing the `<iohw.h>` 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 *I/O-register designators* which support the relevant I/O access methods for the intended compiler market.
3. Provide a way to select the right I/O function at compile time and generate the right machine code based on the *I/O access properties* related to the *I/O-register designators*.

#### C.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 typically used when generating code for the traditional C memory model.

Selection between different code generation alternatives must be determined solely from the *I/O-register designator* declaration for each I/O register. Whenever possible this access method

selection should be implemented such that it may be determined entirely at compile time, in order to avoid any runtime or machine code overhead.

For a compiler vendor, selection between code generation alternatives can always be implemented by supporting different intrinsic access-specification types and keywords designed specially for the given processor architecture, in addition to the standard types and keywords defined by the language.

Simple `<iohw.h>` 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 E regarding simple macro implementations.

Full featured implementations of `<iohw.h>` will require direct compiler support for *I/O-register designators*. See Annex D regarding a generic *I/O-register designator* descriptor.

## C.2 Overview of I/O Hardware Connection Options

The various ways an I/O register can be connected to processor hardware are determined primarily 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 device.
3. The bit width of the processor-bus.

### C.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 device 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 device addressing overview<sup>2</sup>**

| Logical I/O register widths | I/O device bus widths |         |                   |          |                   |         |                   |         |
|-----------------------------|-----------------------|---------|-------------------|----------|-------------------|---------|-------------------|---------|
|                             | 8-bit device bus      |         | 16-bit device bus |          | 32-bit device bus |         | 64-bit device 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               |         |

<sup>2</sup> Note, that this table describes some common bus and register widths for I/O devices. A given platform may use other register and bus widths.

|                                   |         |         |          |          |          |          |        |
|-----------------------------------|---------|---------|----------|----------|----------|----------|--------|
| 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) |         |         |          |          |          |          |        |

### C.2.2 Address Interleave

If the size of the I/O device 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 device is connected to the 32-bit data bus, then three adjacent registers in the I/O device 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 device bus widths | Processor bus widths |              |              |              |
|-----------------------|----------------------|--------------|--------------|--------------|
|                       | 8-bit bus            | 16-bit bus   | 32-bit bus   | 64-bit bus   |
| 8-bit device bus      | Interleave 1         | interleave 2 | Interleave 4 | interleave 8 |
| 16-bit device bus     | n/a                  | interleave 2 | Interleave 4 | interleave 8 |
| 32-bit device bus     | n/a                  | n/a          | Interleave 4 | interleave 8 |
| 64-bit device bus     | n/a                  | n/a          | n/a          | interleave 8 |

(For byte-aligned address ranges)

### C.2.3 I/O Connection Overview:

The two tables above when combined shows all relevant cases for how I/O hardware registers can be connected to a given processor hardware bus, thus:

**Table: Interleave between adjacent I/O registers in buffer**

| I/O Register width | Device 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)

#### C.2.4 Generic buffer index

The interleave distance between two logically adjacent registers in an I/O register array can be calculated from <sup>3</sup>:

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

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

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

Assumptions:

- address range is byte-aligned;
- data bus widths are a whole number of bytes;
- width of the *logical\_IO\_register* is greater than or equal to the width of the *device\_data\_bus*;
- width of the *device\_data\_bus* is less than or equal to the *processor\_data\_bus*.

---

<sup>3</sup> For systems with byte aligned addressing

### C.3 I/O-register designators for different I/O addressing methods

A processor may have more than one addressing range<sup>4</sup>. For each processor addressing range an implementer should consider the following typical addressing methods:

- *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 *I/O-register designator* must contain an address pair (identification of base address register + logical address offset).

The user-defined *base-address* is normally initialised 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 *iohw*.

- *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 only 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 *I/O-register designator* 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 devices that 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 device 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 device processors (e.g. to emulate an external bus). In this case the *I/O-register designator* 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 domain.

---

<sup>4</sup> Processors with a single addressing range only uses memory mapped I/O.

Because of the different number of parameters required and parameter ranges used in an *I/O-register designator*, it is often convenient to define a number of different *I/O-register designator* formats for the different access methods

#### C.4 Atomic operation

It is a requirement of the *<iohw.h>* implementation that in each I/O function a given (partial<sup>5</sup>) 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 recommended that each I/O function in an *<iohw.h>* implementation, be implemented such 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.

#### C.5 Read-modify-write operations and multi-addressing cases.

In general READ-modify-WRITE operations should 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 be completed before the next access operation is initiated.

---

<sup>5</sup> A 32 bit logical register in a device with an 8-bit data bus contains 4 partial I/O registers

## C.6 I/O initialisation

With respect to the standardisation process it is important to make a clear distinction between I/O hardware (device) related initialisation and platform related initialisation. Typically three types of initialisation are related to I/O:

1. I/O device initialisation.
2. I/O access initialisation.
3. I/O base access initialisation.

Here only I/O access initialisation and I/O base access initialisation is relevant for basic I/O hardware addressing.

I/O device initialisation is a natural part of a hardware driver and should always be considered as a part of the I/O driver application itself. This initialisation is done using the standard functions for basic I/O hardware addressing. I/O device initialisation is therefore not a topic for the standardisation process.

I/O access initialisation concerns the initialisation and definition of ***I/O-register designator*** objects. This process is implementation-defined. It depends both on the platform and processor architecture and also on which underlying access methods are supported by the <***iohw.h***

If runtime initialisation is needed this can more efficiently be implemented by splitting the access information in an ***I/O-register designator*** containing only static information and an ***I/O-group designator*** containing a mixture of dynamic and static information. Initialisation at runtime can then be controlled from the user driver level via the I/O hardware interface. The function:

```
io_abs_acquire( _iogroup_designator_ )
```

can be used as a portable way to specify in the source code where and when such initialisation should take place.

I/O base initialisation is used if some of the address information is first available at runtime, or if, for instance, the same I/O driver code needs to service multiple I/O devices of the same type. Initialisation and release of runtime resources related to access bases:

```
iogroup_acquire( _iogroup_designator_ )
iogroup_release( _iogroup_designator_ )
```

If multiple devices are serviced by the same driver code then switching between the devices can be done re-initialisation of the ***I/O-group designator*** information. The function:

```
iogroup_map( _iogroup_designator_ dest, _iogroup_designator_ src)
```

provides a portable way to do this.

With most free-standing environments and embedded systems the platform hardware is well defined, so all ***I/O-group designators*** for I/O registers used by the program can be completely defined at compile time. For such platforms runtime I/O base initialisation is not an issue.

With larger processor systems I/O hardware is often allocated dynamically at runtime. Here the *I/O-group designators* information can only be partly defined at compile time. Some platform software dependent part of it must be initialised at runtime.

When designing the ***I/O-group designator*** object a compiler implementer should therefore make a clear distinction between static information and dynamic information; i.e. what can be defined and initialised at compile time and what must be initialised at runtime.

Depending on the implementation method and depending on whether the ***I/O-group designator*** objects need to contain dynamic information, the ***I/O-group designator*** object may or may not require an instantiation in data memory. Better execution performance can usually be achieved if more of the information is static.

## C.7 Intrinsic Features for I/O Hardware Access

The implementation of I/O hardware access may require for many platforms the use of special machine instructions not otherwise used with the normal C memory model. It is recommended that the compiler vendor provide the necessary intrinsics for operating on any special addressing range supported by the processor.

An I/O hardware implementation should completely encapsulate any intrinsic functionality.

## Annex D

### Generic access\_spec descriptor for I/O hardware addressing

#### D.1 Generic access\_spec descriptor

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

##### D.1.1 Background

Current work has shown that there are three basic requirements which must not be compromised by any standardised 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 standardised 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 fulfil the first two requirements in a consistent way, it should be possible *to refer to a complete access\_spec specification as a single entity*. This is necessary, for instance, to pass access\_spec 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 device 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\_spec for an I/O register. The syntax is a new approach and a super-set solution, intended to replace prior art.

#### D.2 Syntax specification

```
Access specification:
#pragma IODEF( symbolic_port_name, access_method_name, parameter list )
```

```

symbolic_port_name:
    Unique name for the I/O register, defined by the user.

access_method_name:
    Identify how the following list of parameters should be interpreted.

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 device bus type (size and endian of I/O device 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 device bus type:
    device8          // register width = device bus width = 8 bit
    device8l         // register width > device bus width, MSB on low address
    device8h         // register width > device bus width, MSB on high address
    device16         // register width = device bus width = 16-bit
    device16l        // register width > device bus width, MSB on low address
    device16h        // register width > device bus width, MSB on high address
    device32         // register width = device bus width = 32 bit
    device32l        // register width > device bus width, MSB on low address
    device32h        // register width > device bus width, MSB on high address
    device64          // register width = device 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)
```

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 specification method must exist for each access method.

The ACCESS\_METHOD\_NAME is an identifier for the parameter set enclosed in the parenthesis. 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\_spec descriptors.

### D.3 Examples of access\_spec descriptors

Below are some examples of access\_spec parameter combinations for different (typical) access methods. (Each pragma specification below is in the source code placed on a single line).

#### Direct addressing:

```
#pragma IODEF( PORT_NAME, MM_DIRECT,  
            type for I/O register value (size of I/O register),  
            access limitation type,  
            I/O register device bus type (size and endian of I/O device bus),  
            primary address constant,  
            processor bus width type  
        )
```

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 device bus width, then the access operation is built from multiple consecutive addressing operations.

#### Based addressing:

```
#pragma IODEF( PORT_NAME, MM_BASED,  
            type for I/O register value (size of I/O register),  
            access limitation type,  
            I/O register device bus type (size and endian of I/O device bus),  
            primary address constant,  
            processor bus width type,  
            base variable  
        )
```

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 device bus width, then the access operation is built from multiple consecutive addressing operations.

#### Indexed-bus addressing:

```
#pragma IODEF( PORTNAME, MM_INDEXED,
    type for I/O register value (size of I/O register) ,
    access limitation type ,
    I/O register device bus type (size and endian of I/O device bus),
    primary address constant,
    processor bus width type,
    secondary address parameter
)
```

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 devices using a multiplexed address/data bus interface to a processor system having a non-multiplexed interface.

#### Device driver addressing:

```
#pragma IODEF( PORT_NAME, MM_DEVICE_DRIVER,
    type for I/O register value (size of I/O register) ,
    access limitation type ,
    I/O register device bus type (size and endian of I/O device bus),
    primary address constant,
    processor bus width type,
    name of driver function for register write,
    name of driver function for register read
)
```

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 device bus width, then the access operation is built from multiple consecutive addressing operations. (Alternatively, the I/O register device bus type, processor bus width type and the primary address could be transferred to the driver functions.)

#### Direct bit addressing:

```
#pragma IODEF( PORT_NAME, MM_BIT_DIRECT,
    type for I/O register value (size of I/O register),
    access limitation type,
    I/O register device bus type (size and endian of I/O device bus),
    primary address constant,
    processor bus width type,
    bit location in register constant,
)
```

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

Examples:

```
#pragma IODEF( MYPORT, MM_DIRECT, uint8_t,rw_t,device8,0x3000,bw8)
uint8_t a = iord(MYPORT,0xAA); // Read single register
```

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

```
#pragma IODEF( PORTA, MM_DIRECT, uint16_t,wo_t,device81,0x200,bw16)
iowr(PORTA,0xAA); // Write single register
```

PORTA is a 16-bit write-only register, located in a device with an 8-bit data bus (with MSB register part located at the lowest address), where the device 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
#pragma IODEF(DRVREG, MM_DEVICE_DRIVER,
              uint8_t,rmw_t,device8,0xA,my_wr_drv,my_rd_drv)

// 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)
```

## D.4 Parsing

The access specifications are parsed at compile time.

If the symbolic port name is used directly in iord(..) / iowr(..) / etc. functions, the code can be completely optimised at compile time: all information for doing this is available to the compiler at that stage. Based on the combined parameter set, 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_spec` object is needed. This will fulfil the third of the primary requirements in D.1.1 (no-overhead solution).

Example:

```
#pragma IODEF(MY_PORT1, MM_DIRECT, uint16_t,rmw,device8l,0x3456,bw16)
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_spec` object must be instantiated in memory; (slower) generic functions are invoked by the `iord(..)/iowr(..)/etc.` functions. In this case, the `access_spec` parameter is mostly evaluated at runtime. (This approach is similar to the one used for `extern inline` functions in C)

Example:

```
#pragma IODEF( MY_PORT1, MM_DIRECT, uint16_t,rmw_t,device8l,0x3456,bw16)
#pragma IODEF( MY_PORT2, MM_DIRECT, uint16_t,ro_t,device16l,0x7890,bw16)

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);
```

## D.5 Comments on syntax notation

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_spec` definition to new hardware. Maintenance becomes much simpler.

## Annex E

### **Migration path for <iohw.h> implementations.**

#### **E.1 Migration path for <iohw.h> implementations**

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

#### **E.2 <iohw.h> implementation based on C macros**

This chapter illustrates a generic and flexible implementation technique for creating efficient <iohw.h> header implementations based on C macros. This can be done in a relatively straightforward manner common for all processor architectures.

##### **E.2.1 The access specification method**

The generic syntax specification in annex D.2 defines a number of individual access specification parameters which are combined to form the I/O-register designator for a given I/O register. A similar approach is used with this implementation method except that an access type for a given I/O register must be defined by concatenation of the names for the access parameters. For example:

```
#define portname_TYPE <bus>_<method>_<size>_<device bus>_<limitations>
                  MM   DIRECT    8    DEVICE8      RO
                  IO   DIRECT_BUF 16   DEVICE8L     WO
                  INDEXED    32   DEVICE8H     RW
                  INDEXED_BUF        DEVICE16     RMW
                  BASED          DEVICE16L
                  BASED_BUF        DEVICE16H
                  DRIVER          DEVICE32
```

Any additional access parameters required by the given <bus>\_<method> access method must be defined separately in a similar manner:

```
#define portname_<parameter_name>
```

Example:

The full I/O-register designator for direct memory mapped access to a 16-bit write-only register in an 8-bit device consists of two definitions, the access type and the address location:

```
#define PORTA_TYPE      MM_DIRECT_16_DEVICE8L_WO /* PORTA access type */
#define PORTA_ADDR       0x12000  /* PORTA address */
```

### E.2.2 An <iohw.h> implementation technique

The <iohw.h> header can be implemented using a technique called macro specialisation.

The macro specialisation technique uses the following implementation procedure. The I/O access function macros *i0xx(..)* undergo a number of nested macro expansions until it ends in either a special macro for the given access operation, or in a diagnostic message. A diagnostic message can occur if either the specified access method is not supported by the implementation, or if an illegal I/O operation is detected in the source code.

The implementation procedure (without detection of access limitations) follow these steps:

- 1 Define macros which translate *i0xx(portname)* to *portname\_TYPE* and adds the operation type. This defines the access methods.
- 2 Translate the access methods to specialised macro names.
- 3 Define code generation all for the access types and operations in specialised macros.

If more informative error diagnostics and detection of access limitations is wanted an extra expansion level must be used:

- 1 Define macros which translate *i0xx(portname)* to *portname\_TYPE* and adds the operation type. This defines the access methods.
- 2a Translate the access methods to a specialised access operation names.
- 2b Produce informative diagnostic for illegal access operations and translate legal operations to specialised macro names.
- 3 Define code generation for all the access types and operations in specialised macros.

### E.2.3 Features

The <iohw.h> implementation technique proposed has the following *advantages*:

- The specification of an I/O register only requires few source lines pr register.
- The specification syntax is similar across different processor architectures.
- The specification syntax is uniform across compiler implementations.
- Only the code generation macros in step 3 may require modifications in order to adapt an existing implementation to fit a new compiler for the same processor.
- If the access methods are the same and new the processor architecture is similar only the code generation macros in step 3 may require adjustments to the new processor architecture.
- New access methods can be added (or deleted) to suit a particular execution environment or market segment without interaction to the other access method implementations.
- Each access method can be optimised individually for maximum performance with respect to execution speed and memory foot print. For instance by use of compiler intrinsic features.

The <iohw.h> implementation technique proposed has the following *disadvantages*:

- Addition of read/write limitation detection tends to lead to code bloat.

#### E.2.4 The <iohw.h> header

The implementation header below implements the following typical access methods to illustrate the implementation principle:

- 8-bit register and 8-bit register buffer in a memory mapped device.
- 16-bit register and 16-bit register buffer in a memory mapped device
- 16-bit register and 16-bit register buffer in a 8-bit memory mapped device
- 8-bit register and 8-bit register buffer in a I/O mapped device
- 8-bit register and 8-bit register buffer in a device on an I/O mapped indexed bus.

The example assumes the processor hardware two addressing ranges, a 16-bit wide addressing range (memory mapped devices) and an 8-bit wide addressing range (I/O mapped devices).

```
***** Start of IOHW *****
#ifndef IOHW_H
#define IOHW_H
#include <stdint.h> /* uintN_t types */

// Define standard function macros for I/O hardware access.
// Translates symbolic I/O register name to an access type
#define iord(NAME) NAME##_TYPE(RD,NAME,0)
#define iowr(NAME,PVAL) NAME##_TYPE(WR,NAME,(PVAL))
#define ioor(NAME,PVAL) NAME##_TYPE(OR,NAME,(PVAL))
#define ioand(NAME,PVAL) NAME##_TYPE(AND,NAME,(PVAL))
#define ioxor(NAME,PVAL) NAME##_TYPE(XOR,NAME,(PVAL))

#define iordbuf(NAME,INDEX) NAME##_TYPE(RD,NAME,(INDEX),0)
#define iowrbuf(NAME,INDEX,VAL) NAME##_TYPE(WR,NAME,(INDEX),(VAL))
#define ioorbuf(NAME,INDEX,VAL) NAME##_TYPE(OR,NAME,(INDEX),(VAL))
#define ioandbuf(NAME,INDEX,VAL) NAME##_TYPE(AND,NAME,(INDEX),(VAL))
#define ioxorbuf(NAME,INDEX,VAL) NAME##_TYPE(XOR,NAME,(INDEX),(VAL))

#define iogroup_acquire( NAME ) NAME##_INIT
#define iogroup_release( NAME ) NAME##_EXIT
#define iogroup_remap( DNAME,SNAME ) ((DNAME##_ADR) = (SNAME##_ADR))

**** Translate access type for register to specialised macros for the operations ****
// Also resolve most address, index and interleave calculations here, this
// enable single register access and buffer access can share code generation macros.
// Unsupported access methods, that is methods not defined here, will results in a
// compile time error (undefined symbol)

// Memory mapped I/O buffer and register access 8-bit
#define MM_DIR_BUF_8_DEV8( OPR,NAME,INDEX,VAL ) \
    MM_DIR_8_DEV8_##OPR( NAME##_ADR+(INDEX)*MM_INTL,(VAL) )
#define MM_DIR_8_DEV8( OPR,NAME,VAL ) MM_DIR_8_DEV8_##OPR( NAME##_ADR,(VAL) )

// Memory mapped I/O buffer and register access 16-bit
#define MM_DIR_BUF_16_DEV16(OPR,NAME,INDEX,VAL) \
    MM_DIR_16_DEV16_##OPR( NAME##_ADR+(INDEX)*MM_INTL,(VAL) )
#define MM_DIR_16_DEV16( OPR,NAME,VAL ) MM_DIR_16_DEV16_##OPR( NAME##_ADR,(VAL) )

// Memory mapped I/O buffer and register access 16-bit register in 8-bit device
#define MM_DIR_BUF_16_DEV8L( OPR,NAME,INDEX,VAL ) \
    MM_DIR_16_DEV8L_##OPR( NAME##_ADR+(INDEX)*MM_INTL*2,(VAL),MM_INTL )
```

```

#define MM_DIR_16_DEV8L(OPR,NAME,VAL) MM_DIR_16_DEV8L##OPR( NAME##_ADR,(VAL),MM_INTL )

// I/O indexed bus mapped 8-bit buffer and buffer
#define IO_DIR_BUF_8_DEV8(OPR,NAME,INDEX,VAL) \
    IO_DIR_8_DEV8##OPR( NAME##_ADR+(INDEX)*IO_INTL,(VAL) )
#define IO_DIR_8_DEV8(OPR,NAME,VAL) IO_DIR_8_DEV8##OPR( NAME##_ADR,(VAL) )

// I/O indexed bus mapped 8-bit register and buffer
#define IO_IDX_8_DEV8(OPR,NAME,VAL) \
    IO_IDX_8_DEV8##OPR(NAME##_ADR,(VAL),NAME##_SUBADR,IO_INTL)
#define IO_IDX_BUF_8_DEV8(OPR,NAME,INDEX,VAL) \
    IO_IDX_8_DEV8##OPR(NAME##_ADR+(INDEX)*IO_INTL,(VAL),NAME##_SUBADR,IO_INTL)

/* Add access types for other access methods to be supported here */

//***** Some access support macros and intrinsic features *****

#define MM_ACCESS( TYPE, lADR ) (*((TYPE volatile *) (lADR)))
#define IO_INP(a) _inp((unsigned short)(a))
#define IO_OUTP(a,b) _outp((unsigned short)(a),(unsigned char)(b))

#define MM_INTL 2 /* Interleave factor for 16-bit memory mapped bus (fixed) */
#define IO_INTL 1 /* Interleave factor for 8-bit I/O bus (fixed here) */

typedef union { // This compiler uses byte alignment so we can use
    uint16_t w; // a union for fast 8/16-bit conversion
    struct {
        uint8_t b0; // LSB byte
        uint8_t b1; // MSB byte
    }b;
} iohw_union16;

//***** Start of Code generation macros *****
/* MM_DIR_8_DEV8 */
#define MM_DIR_8_DEV8_RD(ADR,VAL) (MM_ACCESS(uint8_t,ADR))
#define MM_DIR_8_DEV8_WR(ADR,VAL) (MM_ACCESS(uint8_t,ADR) = (uint8_t)VAL)
#define MM_DIR_8_DEV8_OR(ADR,VAL) (MM_ACCESS(uint8_t,ADR) |= (uint8_t)VAL)
#define MM_DIR_8_DEV8_AND(ADR,VAL) (MM_ACCESS(uint8_t,ADR) &= (uint8_t)VAL)
#define MM_DIR_8_DEV8_XOR(ADR,VAL) (MM_ACCESS(uint8_t,ADR) ^= (uint8_t)VAL)

/* MM_DIR_16_DEV16 */
#define MM_DIR_16_DEV16_RD(ADR,VAL) (MM_ACCESS(uint16_t,ADR))
#define MM_DIR_16_DEV16_WR(ADR,VAL) (MM_ACCESS(uint16_t,ADR) = (uint16_t)VAL)
#define MM_DIR_16_DEV16_OR(ADR,VAL) (MM_ACCESS(uint16_t,ADR) |= (uint16_t)VAL)
#define MM_DIR_16_DEV16_AND(ADR,VAL) (MM_ACCESS(uint16_t,ADR) &= (uint16_t)VAL)
#define MM_DIR_16_DEV16_XOR(ADR,VAL) (MM_ACCESS(uint16_t,ADR) ^= (uint16_t)VAL)

/* MM_DIR_16_DEV8L */
#define MM_DIR_16_DEV8L_RD(ADR,VAL,INTL) (MM_ACCESS(uint8_t,ADR) * 256 + \
   MM_ACCESS(uint8_t,ADR + INTL))
#define MM_DIR_16_DEV8L_WR(ADR,VAL,INTL) { \
    iohw_union16 temp; \
    temp.w = (uint16_t)VAL; /* Rule C.4 */ \
    MM_ACCESS(uint8_t,ADR) = temp.b.b1; \
    MM_ACCESS(uint8_t,ADR+INTL) = temp.b.b0; \
}
#define MM_DIR_16_DEV8L_OR(ADR,VAL,INTL) MM_DIR_16_8L_OPR(ADR,VAL,INTL,|)
#define MM_DIR_16_DEV8L_AND(ADR,VAL,INTL) MM_DIR_16_8L_OPR(ADR,VAL,INTL,&)
#define MM_DIR_16_DEV8L_XOR(ADR,VAL,INTL) MM_DIR_16_8L_OPR(ADR,VAL,INTL,^)
#define MM_DIR_16_8L_OPR(ADR,VAL,INTL,OPR) { /* Common for | & ^ */ \

```

```

iohw_union16 temp; \
temp.w = (uint16_t)VAL; /* Rule C.4 */ \
temp.b.b1 OPR##= MM_ACCESS(uint8_t,ADR); /* Rule C.5 */ \
temp.b.b0 OPR##= MM_ACCESS(uint8_t,ADR+INTL); \
MM_ACCESS(uint8_t,ADR)      = temp.b.b1; \
MM_ACCESS(uint8_t,ADR+INTL) = temp.b.b0; \
}

/* IO_DIR_8_DEV8 */
#define IO_DIR_8_DEV8_RD(ADR,VAL)  (IO_INP(ADR))
#define IO_DIR_8_DEV8_WR(ADR,VAL)  (IO_OUTP(ADR,VAL))
#define IO_DIR_8_DEV8_OR(ADR,VAL)  (IO_OUTP(ADR,VAL | IO_INP(ADR)))
#define IO_DIR_8_DEV8_AND(ADR,VAL) (IO_OUTP(ADR,VAL & IO_INP(ADR)))
#define IO_DIR_8_DEV8_XOR(ADR,VAL) (IO_OUTP(ADR,VAL ^ IO_INP(ADR)))

/* IO_INDEXED_8_DEV8 */
#define IO_IDX_8_DEV8_RD(ADR,VAL,SUBADR,INTL) (IO_OUTP(ADR,SUBADR),IO_INP(ADR+INTL))
#define IO_IDX_8_DEV8_WR(ADR,VAL,SUBADR,INTL) (IO_OUTP(ADR,SUBADR),IO_OUTP(ADR+INTL,VAL))
#define IO_IDX_8_DEV8_OR(ADR,VAL,SUBADR,INTL) IO_IDX_8_OPR(ADR,VAL,SUBADR,INTL,|)
#define IO_IDX_8_DEV8_AND(ADR,VAL,SUBADR,INTL) IO_IDX_8_OPR(ADR,VAL,SUBADR,INTL,&)
#define IO_IDX_8_DEV8_XOR(ADR,VAL,SUBADR,INTL) IO_IDX_8_OPR(ADR,VAL,SUBADR,INTL,^)
#define IO_IDX_8_OPR(ADR,VAL,SUBADR,INTL,OPR) { /* common for | & ^ */ \
    unsigned char tmp = VAL; /* Rule C.4 */ \
    IO_OUTP(ADR,SUBADR); \
    tmp OPR##= IO_INP(ADR+INTL); \
    IO_OUTP(ADR,SUBADR); \
    IO_OUTP(ADR+INTL,tmp); \
}

/* Add code generation macros for other supported access methods here */

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

```

## E.2.5 The users I/O-register designator definitions

For each I/O-register designator (each symbolic name) a complete definition of the access method must be created. With this I/O hardware implementation the user must define the `access_type` and any address information.

These platform dependent I/O register definitions are normally placed in a separate header file. Here called "**iohw\_my\_hardware**".

```

//***** Start of user I/O register definitions (IOHW_MY_HARDWARE) *****
#ifndef IOHW_MY_HARDWARE
#define IOHW_MY_HARDWARE

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

#define MYPOR1_TYPE       MM_DIR_8_DEV8        // 8-bit register in 8-bit device,
#define MYPOR1_ADR        0xc0000             // memory mapped, use direct access

#define MYPOR2_TYPE       MM_DIR_16_DEV16     // 16-bit register in 16-bit device,
#define MYPOR2_ADR        0xc8000             // memory mapped, use direct access

```

```
#define MYPORT3_TYPE      MM_DIR_16_DEV8L      // 16-bit register in 8-bit device,  
#define MYPORT3_ADR       0xc8040           // memory mapped, use direct access  
  
#define MYPORT4_TYPE      IO_DIR_8_DEV8       // 8-bit register in 8-bit device,  
#define MYPORT4_ADR       0x2345            // I/O bus mapped, use direct access  
  
#define MYPORT5_TYPE      IO_IDX_8_DEV8       // 8-bit register in 8-bit device,  
#define MYPORT5_ADR       0x2345            // I/O indexed bus mapped, use indexed access  
#define MYPORT5_SUBADR   0x56  
  
#define MYPORT6_TYPE      MM_DIR_BUF_8_DEV8    // 8-bit register buffer in 8-bit device,  
#define MYPORT6_ADR       0xb0000           // memory mapped, use direct access  
  
#define MYPORT7_TYPE      MM_DIR_BUF_16_DEV16   // 16-bit register buffer in 16-bit device,  
#define MYPORT7_ADR       0xb8000           // memory mapped, use direct access  
  
#define MYPORT8_TYPE      MM_DIR_BUF_16_DEV8L    // 16-bit register buffer in 8-bit device,  
#define MYPORT8_ADR       0xb4000           // memory mapped, use direct access  
  
#define MYPORT9_TYPE      IO_DIR_BUF_8_DEV8    // 8-bit register buffer in 8-bit device,  
#define MYPORT9_ADR       0x2345            // I/O bus mapped, use direct access  
  
#define MYPORT10_TYPE     IO_IDX_BUF_8_DEV8    // 8-bit register buffer in 8-bit device,  
#define MYPORT10_ADR      0x2345            // I/O indexed bus mapped, use indexed access  
#define MYPORT10_SUBADR  0x56  
  
#endif
```

### E.2.6 The driver function

The driver function should include `<iohw.h>` and the user I/O register definitions for the target system "`iohw_my_hardware`". The example below tests some operations on the previous I/O register definitions.

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

uint8_t cdat;
uint16_t idat;

void my_test_driver (void)
{
    iogroup_acquire(MYPORTS);

    cdat = iord(MYPORT1);           // 8-bit memory mapped register
    iowr(MYPORT1, 0x12);

    iowr(MYPORT2,idat);           // 16-bit memory mapped register
    ioor(MYPORT3, 0x2334);         // 16-bit memory mapped register in 8-bit chip

    ioand(MYPORT4,0x34);          // 8-bit I/O mapped register
    ioxor(MYPORT5,0xf0);          // 8-bit I/O mapped register on indexed bus

    cdat = iordbuf(MYPORT6,20);    // 8 bit memory mapped register

    iowrbuf(MYPORT7,43,0x3458);   // 16-bit memory mapped register
    ioorbuf(MYPORT8,43,idat);     // 16-bit memory mapped register in 8 bit chip

    ioandbuf(MYPORT9,43,0x02);    // 8 bit I/O mapped register
    ioxorbuf(MYPORT10,43,0x12);   // 8 bit I/O mapped register on indexed bus

    iogroup_release(MYPORTS);
}
```

## Annex F

### Functionality not included in this Technical Report.

#### F.1 Circular buffers

The concept of circular buffers is widely used within the signal processing community. An example of the use of the concept of circular buffers is in a FIR filter, where it is used to reduce the number of memory accesses. The functionality of a FIR filter can be described in this way with current C:

```

int x[N+1]; // data values
int h[N+1]; // coefficient values
long int accu = 0;

x[0] = new_value;

accu = x[N] * h[N];

for(i=N-1; i>0; i--)
{
    accu += (long int) x[i] * h[i];
    x[i]= x[i-1];
}

```

The data value copy in the last statement in the for loop would be unnecessary, if the concept of a circular buffer was employed here, reducing the number of memory accesses. Many digital signal processors have direct support in their addressing hardware to provide zero-overhead circular addressing. Zero-overhead means here that calculating the address for an access to a circular buffer can be done in the same time as performing a regular address calculation, including the wrap-around check and, if necessary, the execution of the wrap-around. However there are often many restrictions on how hardware supported circular addressing can be used. E.g., only address increments by one are allowed in some implementations, and there may be requirements to the size and/or alignment of the buffer.

Since the functional specifications of the support for circular addressing in various processors is so diverse, it is difficult to define an abstract model that can be used in a natural manner in the C language, and that also can be translated efficiently for the various hardware paradigms. Therefore, in this Technical Report no proposals are made for language extensions to support circular buffers. Should, in the future, a single approach towards circular addressing become dominant in the market, then an appropriate C language construct could be defined.

Some current approaches to circular addressing are given below.

- Add a new keyword (for instance, **circ**) to the C language, that allows a programmer to indicate that an array or pointer with this qualifier is to be accessed with circular addressing.
- Another solution is to define a new library function or macro, CIRC(), which could be used in the following manner:

```
int *p;
```

```
p = CIRC(p+1, /* array info */);
    // this does an increment by one of p
```

Array info in this example covers the starting address and end address of the address range where circular addressing is desired. A compiler for an architecture that has direct hardware support for circular addressing is then free to optimise this function call away, and exploit the capabilities of the hardware.

- In the current C language there is provision to specify circular buffers, however only when using array index notation:

```
accum += (long int) x[i % N]*h[i];
```

It is possible for a clever optimiser to recognise that this in fact is a circular buffer and exploit the hardware support for this. This has the advantage that the use of circular buffers is already possible within the current C language, but it requires the programmer to use array indices rather than pointers. Furthermore it is not possible to specify any alignment constraint on the allocated buffer, which might be necessary for the underlying hardware implementation.

No preferred solution is specified here.

## F.2 Complex data types

In this Technical Report no complex fixed-point data types are been defined. However in the C language, complex data types are already existing for floating-point numbers. As **fract** and **accum** types can be viewed upon as an alternative to floating-point numbers in some applications it is worthwhile considering extending the definition of complex types in C to include fract and accum bases. It will be beneficial for the user community to standardise such data types as they have a clear usage in an area like communications signal processing.

## F.3 Consideration of BCD data types for Embedded Systems

It was briefly considered to include some form of Binary Coded Decimal (BCD) as part of the Embedded Systems C Technical Report. BCD types have been frequently proposed for embedded systems and financial applications. As the area of application for BCD types is too diverse, and since in this version of this Technical Report only binary types are considered, it was decided not to include BCD types.

## F.4 Modwrap overflow

Next to the saturated overflow behaviour, sometimes another overflow behaviour is desired: modular wrap-around. This exploits an important property of 2's complement representation of fixed-point data types, and is used in e.g. sums-of-products calculation, where the sum of a number of the individual products may overflow, but the full sum does not overflow. In this case modular wrap-around on overflow may be used.

Modular wrap-around overflow handling can be defined as follows:

For unsigned fixed-point types, the source value is replaced by a value within the range of the fixed-point type that is congruent (in the mathematical sense) to the source value modulo  $2^N$ , where N is the number of integral bits in the fixed-point type. (For example, for unsigned **fract** types, N equals 0, and the source value is replaced by a value between 0 and 1 that is congruent to the source value modulo 1.) For signed fixed-point types, the source value is replaced by a value within the fixed-point range that is congruent to the source value modulo  $2^{(N+1)}$ , where N again is the number of integral bits in the fixed-point type. (In either case, the effect is to discard all bits above the most significant bit of the fixed-point format.)

At some point, inclusion in this TR of an additional type qualifier (**\_Modwrap**) to specify modular wrap-around on overflow for the fixed-point types was considered. However, seeing its marginal use and the added complexity of an extra type qualifier it was decided not to include this functionality in the current text.

Implementations are however free to provide mechanisms to support the modwrap functionality.

## Annex G

### Compatibility and Migration issues.

#### G.1 Compatibility with C++

It is recognised that the functionality, described in the Technical Report, might also be useful in environments where C++ is the dominant programming language. At the same time it is recognised that the preferred C++ syntax and mechanisms to incorporate the described functionality is different.

In programming environments where the same code is envisioned to be used for both C and C++ it is recommended to use a programming style that can easily support programming paradigms from both communities. Unfortunately this programming style will neither be a C style or a C++ style. In the absence of a formally agreed approach to this problem, this section gives some guidelines on how to use the functionality specified in this Technical Report in a C++ environment.

##### G.1.1 Fixed-point Arithmetic

Implementation of fixed-point arithmetic in C++ is simple and straightforward. Each fixed-point type (such as `sat long fixed`) can be provided as a C++ class (such as class `sat_long_fixed`) which provides its own set of overloaded operators. The fixed-point type-balancing rules can be implemented by providing several dozen specific operand types for each overloaded operator. The constructor syntax of C++ provides a method for conversion from integer and floating types to fixed-point types.

##### G.1.2 Multiple Address Spaces Support

Named address space support could be added to C++ only as an extension; it would not be implementable within the class library. Such an extension would not conflict with existing features of C++.

##### G.1.3 Basic I/O Hardware Addressing

Regarding the C++ compatibility of iohw.h, refer to ISO/IEC TR 18015 (C++ Performance). That Technical Report contains extensive discussion of methods for implementing the iohw capability in C++, making use of the templates capability of C++.