

Programming languages — C — Extensions to support embedded processors

ICS 35.060

National foreword

This Published Document reproduces verbatim ISO/IEC TR 18037:2004.
The UK participation in its preparation was entrusted to Technical Committee IST/5, Programming languages, which has the responsibility to:

- aid enquirers to understand the text;
- present to the responsible international/European committee any enquiries on the interpretation, or proposals for change, and keep the UK interests informed;
- monitor related international and European developments and promulgate them in the UK.

A list of organizations represented on this committee can be obtained on request to its secretary.

Cross-references

The British Standards which implement international publications referred to in this document may be found in the *BSI Catalogue* under the section entitled “International Standards Correspondence Index”, or by using the “Search” facility of the *BSI Electronic Catalogue* or of British Standards Online.

Summary of pages

This document comprises a front cover, an inside front cover, the ISO/IEC TR title page, pages ii to vii, a blank page, pages 1 to 96, an inside back cover and a back cover.

The BSI copyright notice displayed in this document indicates when the document was last issued.

Amendments issued since publication

Amd. No.	Date	Comments

This Published Document was published under the authority of the Standards Policy and Strategy Committee on 10 November 2004

© BSI 10 November 2004

ISBN 0 580 44714 6

TECHNICAL REPORT

**ISO/IEC
TR
18037**

First edition
2004-07-15

Programming languages — C — Extensions to support embedded processors

*Langages de programmation — C — Extensions pour supporter les
processeurs intégrés*



Contents

Page

FOREWORD	VI
INTRODUCTION	VII
1 SCOPE	1
2 REFERENCES	1
3 CONFORMANCE	1
4 FIXED-POINT ARITHMETIC	2
4.1 Overview and principles of the fixed-point data types	2
4.1.1 The data types	2
4.1.2 Spelling of the new keywords	3
4.1.3 Overflow and Rounding	4
4.1.4 Type conversion, usual arithmetic conversions	5
4.1.5 Fixed-point constants	6
4.1.6 Operations involving fixed-point types	7
4.1.7 Fixed-point functions	9
4.1.8 Fixed-point definitions <stdint.h>	11
4.1.9 Formatted I/O functions for fixed-point arguments	11
4.2 Detailed changes to ISO/IEC 9899:1999	12
5 NAMED ADDRESS SPACES AND NAMED-REGISTER STORAGE CLASSES	37
5.1 Overview and principles of named address spaces	37
5.1.1 Additional address spaces	37
5.1.2 Address-space type qualifiers	37
5.1.3 Address space nesting and rules for pointers	38
5.1.4 Standard library support	39
5.2 Overview and principles of named-register storage classes	39
5.2.1 Access to machine registers	39
5.2.2 Named-register storage-class specifiers	39
5.2.3 Ensuring correct side effects via objects allocated in registers	41
5.2.4 Relationship between named registers and I/O-register designators	41
5.3 Detailed changes to ISO/IEC 9899:1999	41
6 BASIC I/O HARDWARE ADDRESSING	49
6.1 Rationale	49
6.1.1 Basic Standardization Objectives	49

6.2 Terminology	49
6.3 Basic I/O Hardware addressing header <iohw.h>	51
6.3.1 Standardization principles	51
6.3.2 The abstract model.....	52
6.4 Specifying I/O registers	54
6.4.1 I/O-register designators.....	54
6.4.2 Accesses to individual I/O registers	54
6.4.3 I/O register buffers.....	55
6.4.4 I/O groups.....	56
6.4.5 Direct and indirect designators.....	57
6.4.6 Operations on I/O groups	57
6.5 Detailed changes to ISO/IEC 9899:1999	58
ANNEX A - FIXED-POINT ARITHMETIC	65
A.1 Fixed-point datatypes	65
A.1.1 Introduction.....	65
A.2. Number of data bits in _Fract versus _Accum	68
A.3 Possible Data Type Implementations.....	69
A.4 Overflow and Rounding.....	70
A.5 Type conversions, usual arithmetic conversions	71
A.6 Operations involving fixed-point types	72
A.7 Exception for 1 and –1 Multiplication Results	72
A.8 Linguistic Variables and unsigned _Fract: an example of unsigned fixed-point	73
ANNEX B - NAMED ADDRESS SPACES AND NAMED-REGISTER STORAGE CLASSES	74
B.1 Embedded systems extended memory support.....	74
B.1.1 Modifiers for named address spaces	74
B.1.2 Application-defined multiple address space support.....	75
B.1.3 I/O register definition for intrinsic or user defined address spaces	76
ANNEX C - IMPLEMENTING THE <IOHW.H> HEADER.....	78
C.1 General.....	78
C.1.1 Recommended steps	78
C.1.2 Compiler considerations.....	78
C.2 Overview of I/O Hardware Connection Options	79
C.2.1 Multi-Addressing and I/O Register Endianness	79
C.2.2 Address Interleaving.....	80
C.2.3 I/O Connection Overview:	80
C.2.4 Generic buffer index	81

C.3.	I/O-register designators for different I/O addressing methods	82
C.4	Atomic operation	83
C.5	Read-modify-write operations and multi-addressing cases.	83
C.6	I/O initialization	84
C.7	Intrinsic Features for I/O Hardware Access	85
ANNEX D - MIGRATION PATH FOR <IOHW.H> IMPLEMENTATIONS		86
D.1	Migration path for <iohw.h> implementations	86
D.2	<iohw.h> implementation based on C macros	86
D.2.1	The access specification method	86
D.2.2	An <iohw.h> implementation technique	87
D.2.3	Features	87
D.2.4	The <iohw.h> header	88
D.2.5	The user's I/O-register designator definitions	91
D.2.6	The driver function.....	92
ANNEX E - FUNCTIONALITY NOT INCLUDED IN THIS TECHNICAL REPORT		93
E.1	Circular buffers	93
E.2	Complex data types.....	94
E.3	Consideration of BCD data types for Embedded Systems	94
E.4	Modwrap overflow	94
ANNEX F - C++ COMPATIBILITY AND MIGRATION ISSUES.....		96
F.1	Fixed-point Arithmetic.....	96
F.2	Multiple Address Spaces Support	96
F.3	Basic I/O Hardware Addressing	96

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

In exceptional circumstances, the joint technical committee may propose the publication of a Technical Report of one of the following types:

- type 1, when the required support cannot be obtained for the publication of an International Standard, despite repeated efforts;
- type 2, when the subject is still under technical development or where for any other reason there is the future but not immediate possibility of an agreement on an International Standard;
- type 3, when the joint technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example).

Technical Reports of types 1 and 2 are subject to review within three years of publication, to decide whether they can be transformed into International Standards. Technical Reports of type 3 do not necessarily have to be reviewed until the data they provide are considered to be no longer valid or useful.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TR 18037, which is a Technical Report of type 2, was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

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 become more complex (and hence are difficult to maintain in assembly language), and processor models for embedded systems have a decreasing lifespan (which implies more frequent re-adapting of applications to new instruction sets). 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 analyze 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, embedded processors are often equipped with special hardware for fixed-point data. The C language (as defined in ISO/IEC 9899:1999) does not provide support for fixed-point arithmetic operations, currently leaving programmers with no option but to handcraft most of their algorithms in assembly language. This Technical Report specifies a fixed-point data type for C, definable in a range of precision and saturation options. Optimizing C compilers can generate highly efficient code for fixed-point data as easily as for integer and floating-point data.

Many embedded processors have multiple distinct banks of memory and require that data be grouped in different banks to achieve maximum performance. 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, optimizing compilers can utilize the ability of processors that support multiple address spaces, for instance, to read data from two separate memories in a single cycle to maximize 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.

Programming languages — C — Extensions to support embedded processors

1 Scope

This Technical Report specifies a series of extensions of the programming language C (as specified by ISO/IEC 9899:1999) to support features commonly found in embedded processors. It deals with the following topics: extensions to support fixed-point arithmetic, named address spaces, and basic I/O hardware addressing.

Each clause in this Technical Report deals with a specific topic. The first subclauses of clauses 4, 5 and 6 contain a technical description of the features of the topic. These subclauses provide an overview but do not contain all the fine details. The last subclause of each clause contains the editorial changes to ISO/IEC 9899:1999 necessary to fully specify the topic in ISO/IEC 9899:1999, and thereby provides a complete definition. Additional explanation and rationale are provided in the Annexes.

2 References

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 9899:1999, *Programming languages – C*

3 Conformance

This Technical Report presents in three separate clauses specifications for three, in principle independent, sets of functionality (clause 4: fixed-point arithmetic, clause 5: named address spaces and named-register storage classes, and clause 6: basic I/O hardware addressing). As this is a Technical Report there are no conformance requirements and implementers are free to select those specifications that they need. However, if functionality is implemented from one of the clauses, implementers are strongly encouraged to implement that clause in full, and not just a part of it.

If, at a later stage, a decision is taken to incorporate some or all of the text of this Technical Report into the C standard, then at that moment the conformance issues with respect to (parts of) this text need to be addressed (conformance with respect to freestanding implementations etc.)

4 Fixed-point arithmetic

4.1 Overview and principles of the fixed-point data types

4.1.1 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:

unsigned short _Fract	unsigned short _Accum
unsigned _Fract	unsigned _Accum
unsigned long _Fract	unsigned long _Accum
signed short _Fract	signed short _Accum
signed _Fract	signed _Accum
signed long _Fract	signed long _Accum

These twelve types are collectively called the *primary fixed-point types*. The fixed-point data types

short _Fract	short _Accum
_Fract	_Accum
long _Fract	long _Accum

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

For each primary fixed-point type there is a corresponding (but different) *saturating fixed-point type*, designated with the type-specifier **_Sat**. The primary fixed-point types and the saturating fixed-point types are collectively called the *fixed-point types*.

An implementation is required to support all above-mentioned twenty-four 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. Each saturating fixed-point type has the same representation and the same rank as its corresponding primary 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:

signed short _Fract	s.7	signed short _Accum	s4.7
signed _Fract	s.15	signed _Accum	s4.15
signed long _Fract	s.23	signed long _Accum	s4.23
unsigned short _Fract	.7	unsigned short _Accum	4.7
unsigned _Fract	.15	unsigned _Accum	4.15
unsigned long _Fract	.23	unsigned long _Accum	4.23

(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 clause 6.2.5 (see clause 4.2 of this Technical Report).

For an example of unsigned fixed-point datatypes see A.8.

4.1.2 Spelling of the new keywords

The natural spelling of the newly introduced keywords **_Fract**, **_Accum** and **_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 are used to define the

natural spellings as aliases, and may be used to define other spellings, 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 F.

4.1.3 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 Annex E.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:

- By using explicit saturating fixed-point types (e.g., **_Sat_Fract**).
- In the absence of an explicit saturating fixed-point type, 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 behavior 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 behavior 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 behavior for those computations where the actual overflow behavior is not relevant; the actual overflow behavior 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 the correct result; often these additional instructions are not needed if the required precision is 2 ulps¹. 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 requirements are allowed (the state of the **FX_FULL_PRECISION** pragma is "off"). For more discussion on this topic see A.4.

Whether rounding is up or down is implementation-defined and may differ for different values and different situations; an implementation may specify that the rounding is indeterminable.

4.1.4 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;
2. if one operand has signed fixed-point type and the other operand has unsigned fixed-point type, then the unsigned fixed-point operand is converted to its corresponding signed fixed-point type and the resulting type is the type of the converted operand;
3. the result type is the type with the highest rank, whereby a fixed-point conversion rank is always greater than an integer conversion rank; if the type of either of the operands is a saturating fixed-point type, the result type shall be the saturating fixed-point type corresponding to the type with the highest rank; the resulting value is converted (taking into account rounding and overflow) to the precision of the resulting type.

¹ unit in the last place: precision up to the last bit

EXAMPLE: in the following code fragment:

```
fract f = 0.25r;
int i = 3;

f = f * i;
```

the variable `f` gets the value 0.75.

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 one of the operands has a floating type and the other operand has a fixed-point type, the fixed-point operand is converted to the floating type in the usual way.

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.

4.1.5 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
hr	short _Fract
uhr	unsigned short _Fract
r	_Fract
ur	unsigned _Fract
lr	long _Fract
ulr	unsigned long _Fract
hk	short _Accum
uhk	unsigned short _Accum
k	_Accum
uk	unsigned _Accum

lk	long _Accum
ulk	unsigned long _Accum

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.

4.1.6 Operations involving fixed-point types

4.1.6.1 Unary operators

4.1.6.1.1 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.

4.1.6.1.2 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**.

4.1.6.2 Binary operators

4.1.6.2.1 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 4.1.4, 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 4.1.3. 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.7.

If the value of the second operand of the $/$ operator is zero, the behavior 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 **idivfx**, where **fx** stands for one of **r**, **lr**, **k**, **lk**, **ur**, **ulr**, **uk** and **ulk**. The **mulifx** functions multiply an integer operand by a fixed-point operand and return an integer value; the **divifx** 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.

4.1.6.2.2 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 must have integer type 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.

4.1.6.2.3 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.

4.1.6.3 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.

4.1.6.4 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.

4.1.7 Fixed-point functions

4.1.7.1 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 result 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.

4.1.7.2 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 result 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 result 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.

4.1.7.3 The fixed-point bit counts 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 result 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.

4.1.7.4 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** (for signed fixed-point types) or **uint_fx_t** (for unsigned fixed-point types), 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 $0.5 * 2^{15} = 0x4000$.

4.1.7.5 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_fx_t** or **uint_fx_t**, the result type is a fixed-point type (corresponding to **fx**).

The bitwise fixed-point to integer conversion functions return a 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.

4.1.7.6 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 4.1.7.1, the fixed-point round functions in 4.1.7.2 and the fixed-point countls functions in 4.1.7.3, a type-generic macro is defined as follows:

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

4.1.7.7 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 result 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.

4.1.8 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.

4.1.9 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 **strtofx** function. The corresponding argument of **fscanf** shall be a pointer to a fixed-point type variable with a type corresponding to **fx**.

4.2 Detailed changes to ISO/IEC 9899:1999

This clause details the modifications to ISO/IEC 9899:1999 needed to incorporate the fixed-point functionality as described in clause 4.1 of this Technical Report. The changes listed in this clause are limited to syntax and semantics; examples, (forward) references and other descriptive information are omitted. The modifications are ordered according to the clauses of ISO/IEC 9899:1999 to which they refer. If a clause of ISO/IEC 9899:1999 is not mentioned, no changes to that clause are needed. New clauses are indicated with **(NEW CLAUSE)**, however resulting changes in the existing numbering are not indicated; the clause number *mm.nna* of new clause indicates that this clause follows immediately clause *mm.nn* at the same level.

Clause 5.2.4.2.3 - Characteristics of fixed-point types (NEW CLAUSE)

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:

<i>s</i>	sign (± 1)
<i>b</i>	base or radix of nominator representation (an integer > 1)
<i>p</i>	precision (the number of base- <i>b</i> digits in the nominator)
<i>n</i>	nominator (nonnegative integer less than <i>b</i> raised to the power <i>p</i>)
<i>f</i>	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 somewhere between the most significant digit and the least significant digit in the nominator, 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 on the operand values according to the operation's usual mathematical definition, and then overflow handling and rounding is performed for the result type.

The fixed-point overflow behavior is either saturation or undefined. 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 most positive value of the fixed-point type. For signed fixed-point types it will be the most negative or most positive value of the fixed-point type.)

Overflow behavior is determined as follows:

- If the result type of an arithmetic operation is a saturating fixed-point type (see clause 6.2.5) the overflow behavior is saturation.
- If the result type is a primary fixed-point type (see clause 6.2.5), 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 behavior 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 behavior 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 behavior for those computations where the actual overflow behavior is not relevant; the actual overflow behavior 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; an implementation may specify that the rounding is indeterminable.

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 7.18a.6.1.

Clause 6.2.5 - Types, add the following new paragraphs after paragraph 9:

There are six *primary signed fixed-point types*, designated as **short _Fract**, **_Fract**, **long _Fract**, **short _Accum**, **_Accum**, and **long _Accum**. For each of the primary signed fixed-point types, there is a corresponding (but different) *primary unsigned fixed-point type* (designated with the keyword **unsigned**) that uses the same amount of storage and has the same alignment requirements as its corresponding signed type. The primary signed fixed-point types and the primary unsigned fixed-point types are collectively called *primary fixed-point types*.

For each of the primary fixed-point types, there is a corresponding (but different) *saturating fixed-point type* (designated with the keyword **_Sat**) that has the same representation and the same alignment requirements as its corresponding type. The primary fixed-point types and the saturating fixed-point types are collectively called *fixed-point types*.

The six types **short _Fract**, **_Fract**, **long _Fract**, **short _Sat _Fract**, **_Sat _Fract**, and **long _Sat _Fract** are collectively called *signed fract types*. The six types **unsigned short _Fract**, **unsigned _Fract**, **unsigned long _Fract**, **_Sat unsigned short _Fract**, **_Sat unsigned _Fract**, and **_Sat unsigned long _Fract** are collectively called *unsigned fract types*. The signed fract types and the unsigned fract types are collectively called *fract types*.

The six types **short _Accum**, **_Accum**, **long _Accum**, **_Sat short Accum**, **_Sat _Accum**, and **_Sat long Accum** are collectively called *signed accum types*. The six types **unsigned short _Accum**, **unsigned _Accum**, **unsigned long _Accum**, **_Sat unsigned short _Accum**, **_Sat unsigned _Accum**, and **_Sat unsigned long _Accum** are collectively called *unsigned accum types*. The signed accum types and the unsigned accum types are collectively called *accum types*.

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

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

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

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

Clause 6.2.6.3 - Fixed-point types (NEW CLAUSE)

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 value bit shall represent a different power of 2 between $2^{(-1)}$ and $2^{(-N)}$, and the sign bit shall represent the value -1; objects with signed fract type shall be capable of representing values from -1 to $1-2^{(-N)}$ using a pure binary representation. For accum types, each value bit shall represent a different power of 2 between $2^{(L-1)}$ and $2^{(L-N)}$, and the sign bit shall represent the value of -2^L ; objects with signed accum 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.

The minimum values for the number of fractional bits and the number of integral bits in the various fixed-point types are specified in 7.18a.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 as, or one more fractional bit than, its corresponding signed fract type.
- When arranged in order of increasing rank (see 6.3.1.3a), 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 (see 6.3.1.3a), 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:

- 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.

Clause 6.3.1.3a - Fixed-point types (NEW CLAUSE)

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.
Each saturating fixed-point type has the same rank as its corresponding primary 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.

Clause 6.3.1.8 - Usual arithmetic conversions, replace second and third sentence of paragraph 1 with

In most cases, the purpose is to determine a *common real type* for the operands and result.
(Exceptions often apply for fixed-point operands.)

Clause 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; the result type shall be the fixed-point type.

Otherwise, if both operands have signed fixed-point types, or if both operands have unsigned fixed-point types, then no conversions are needed; the result type shall be the fixed-point type with the higher fixed-point rank; if either of the operands has a saturating fixed-point type, the result type shall be the saturating fixed-point type corresponding to the fixed-point type with the higher fixed-point rank.

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; the result type shall be the signed fixed-point type with the higher fixed-point rank; if either of the operands has a saturating fixed-point type, the result type shall be the saturating fixed-point type corresponding to the signed fixed-point type with the higher fixed-point rank.

Clause 6.4.1 - Keywords, add the following new keywords:

`_Accum` `_Fract` `_Sat`

Clause 6.4.4 - Constants, change the constraints clause to:

The value of a constant shall be in the range of representable values for its type, with exception for constants of a `fract` type with a value of exactly 1; such a constant shall denote the maximal value for the type.

Clause 6.4.4.2a - Fixed-point constants (NEW CLAUSE)**Syntax***fixed-constant:**decimal-fixed-constant**hexadecimal-fixed-constant**decimal-fixed-constant:**fractional-constant exponent-part_{opt} fixed-suffix**digit-sequence exponent-part_{opt} 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_{opt} fxp-suffix_{opt} fixed-qual**fxp-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 (see 6.4.4.2). If suffixed by the letter **r** or **R**, the constant has a fract type; if suffixed by the letter **k** or **K**, the constant has an accum type; if suffixed by the letter **h** or **H**, the constant has a short fract type or a short accum type.

Clause 6.5.7 - Bitwise shift operands, change the constraints clause as follows:

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

Clause 6.5.7 - Bitwise shift operands, replace second and third sentence of paragraph 4 with:

If **E1** has a fixed-point type, the value of the result is **E1 * 2^{E2}**. If **E1** has an unsigned integer type, the value of the result is **E1 * 2^{E2}**, reduced modulo one more than the maximum value representable

in the result type. If **E1** has a signed integer type and nonnegative value, and $\mathbf{E1} * 2^{\mathbf{E2}}$ is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

Clause 6.5.7 - Bitwise shift operands, replace second and third sentence of paragraph 5 with:

If **E1** has a fixed-point type, the value of the result is $\mathbf{E1} * 2^{(-\mathbf{E2})}$. If **E1** has an unsigned integer type or if **E1** has a signed integer type and a nonnegative value, the value of the result is the integral part of the quotient of $\mathbf{E1} / 2^{\mathbf{E2}}$. If **E1** has a signed integer type and a negative value, the resulting value is implementation-defined.

Clause 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, ...

Clause 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.

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

```
_Fract
_Accum
_Sat
```

Clause 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**

- `_Sat signed short _Fract`, or `_Sat short _Fract`
- `_Sat signed _Fract`, or `_Sat _Fract`
- `_Sat signed long _Fract`, or `_Sat long _Fract`
- `_Sat signed short _Accum`, or `_Sat short _Accum`
- `_Sat signed _Accum`, or `_Sat _Accum`
- `_Sat signed long _Accum`, or `_Sat long _Accum`
- `_Sat unsigned short _Fract`
- `_Sat unsigned _Fract`
- `_Sat unsigned long _Fract`
- `_Sat unsigned short _Accum`
- `_Sat unsigned _Accum`
- `_Sat unsigned long _Accum`

Clause 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**

Clause 7.1.2 - Standard headers, add to paragraph 2:

`<stdfix.h>`

Clause 7.18a - Fixed-point arithmetic <stdfix.h> (NEW CLAUSE)

7.18a.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**.

7.18a.2 Integer types used as return types for the bits conversion functions

The following integer types are introduced as typedefs:

```
int_hr_t
int_r_t
int_lr_t
int_hk_t
int_k_t
int_lk_t
uint_uhr_t
uint_ur_t
uint_ulr_t
uint_uhk_t
uint_uk_t
uint_ulk_t
```

The types **int_fx_t** and **uint_fx_t** are the return types of the corresponding **bitsfx** functions, and are 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.

7.18a.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 clause, as an addition to the limits required by the ISO C standard.

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

The values in the following list 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 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 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 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 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.FFFFFFFCP-1LR // hex constant

- 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