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

Reference number
ISO/IEC TR 18037:2004(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

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-specifier*s **_Fract** and **_Accum**. These *type-specifiers* can be used in combination with the existing *type-specifier*s **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 ulps[1]. 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