ISO/IEC JTC1/SC22 WG14/N854

DSP-C

An extension to ISO/IEC IS 9899:1990

ACE Associated Compiler Experts bv

Release:	9.9
Date:	October 7, 1998
Status:	release
Confidentiality:	public
Reference:	CoSy-8025P-dsp-c

Printed in The Netherlands.

Contents

1 Introduction

1.1 Purpose

This document defines an extension to the ISO/IEC IS 9899:1990 ("ISO C") standard to support the specific hardware features of Digital Signal Processors (DSP's). The most important basic language elements added are a fixedpoint data type (in various forms), memory spaces and circular pointers. Some features of this extension, most notably the memory spaces, may also be applicable to embedded processors which are not DSP's, such as microcontrollers.

The DSP-C extensions to the ISO-C definition as specified in this document combined with the ISO-C definition specify a language ("DSP-C") that is meant to be generic for different DSP's. This means that applications written in DSP-C for use on one DSP should be portable and can be compiled using any compiler supporting this language extension. Applications which rely on implementation defined aspects, such as the size of the various data types, may produce different results when compiled for a different DSP. This is actually no different from the situation which exists for ISO C programs.

1.2 Scope

This document specifies the form and establishes the interpretation of programs written in DSP-C, an extension to the ISO C programming language defined in ISO/IEC IS 9899:1990. It specifies:

- the syntax and constraints of the DSP extensions to ISO C
- the semantic rules for interpreting these extensions
- the representation of input data to be processed by these extensions
- the representation of output data
- the restrictions and limits imposed on a conforming implementation of these extensions

1.3 Organization of the document

This document has the same structure as the ISO/IEC IS 9899:1990 document, and is divided into four major sections:

- this introduction
- the characteristics of environments that translate and execute C programs
- the language syntax, constraints and semantics
- the library facilities

This document is meant as an addition to the ISO/IEC IS 9899:1990 document, and should be read in conjunction with it. Appendices cover fixedpoint data representation and other implementation issues. These appendices are not part of the DSP-C specification.

1.4 Standardization

DSP-C has been defined because of the lack of an open, portable extension to ISO C (as opposed to C++ or a subset thereof). An implementation of DSP-C is currently available to users of the CoSy compilation system, but the language specification is also available to other compiler developers. The current specification is expected to evolve further over time, as implementations for more DSP's lead to new user requirements. DSP-C will be submitted to the relevant ISO standardization committees for inclusion in a future ISO C standard.

Comments on this document, as well as requests for more information about this standard, can be sent by electronic mail to dspc@ace.nl.

1.5 Acknowledgements

This language specification has been made possible by Philips Semiconductors NV. Although it is impossible to mention all the people that have somehow contributed to this specification, the most important contributors have been:
Jan van Dongen
Job Ganzevoort
Jos van der Heijden
Ernst van der Horst
Martien de Jong
Wim Kloosterhuis
Rob Kurver
Martijn de Lange
Hans van Someren
Rob Woudsma

2 Normative references

No additions in this chapter.

3 Definitions and conventions

No additions in this chapter.

4 Compliance

No additions in this chapter.

5 Environment

5.1 Conceptual models

No additions in this section.

5.1.1 Translation environment

No additions in this section.

5.1.1.1 Program structure

No additions in this section.

5.1.1.2 Translation phases

No additions in this section.

5.1.1.3 Diagnostics

No additions in this section.

5.1.2 Execution environments

No additions in this section.

5.1.2.1 Free-standing environment

No additions in this section.

5.1.2.2 Hosted environment

No additions in this section.

5.1.2.3 Program execution

No additions in this section.

5.2 Environmental considerations

No additions in this section.

5.2.1 Character sets

No additions in this section.

5.2.1.1 Trigraph sequences

No additions in this section.

5.2.1.2 Multibyte characters

No additions in this section.

5.2.2 Character display semantics

No additions in this section.

5.2.3 Signals and interrupts

No additions in this section.

5.2.4 Environmental limits

No additions in this section.

5.2.4.1 Translation limits

No additions in this section.

5.2.4.2 Numerical limits

No additions in this section.

5.2.4.3 Fixedpoint limits

New constants are introduced to denote the behavior and limits of fixedpoint 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 limits specified in this section shall be specified in the header file `<fixed.h>`.

See also Appendix A for an explanation of fixedpoint types.

Sizes of fixedpoint types `<fixed.h>`

The values given below shall be replaced by constant expressions suitable for use in `#if` preprocessing directives. Moreover, the following shall be replaced by expressions that have the same type as would an expression that is an object of the corresponding type converted according to the promotion rules. Except for the various `EPSILON` values, their implementation-defined values shall be equal or greater in magnitude (absolute value) to those shown, with the same sign. For the various `EPSILON` values, their implementation-defined values shall be equal or smaller in magnitude to those shown.

- number of bits for object of type `signed short __fixed`

  `SFIXED_BIT 8`

- minimum value for an object of type `signed short __fixed`

  `SFIXED_MIN (-0.5r-0.5r)`

- maximum value for an object of type `signed short __fixed`

  `SFIXED_MAX 0.9921875r`

- the difference between 0.0r and the least value greater than 0.0r that is representable in the `signed short __fixed` type

  `SFIXED_EPSILON 0.0078125r`

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

  `USFIXED_MAX 0.9921875ur`

- the difference between 0.0r and the least value greater than 0.0r that is representable in the `unsigned short __fixed` type

  `USFIXED_EPSILON 0.0078125ur`

- number of bits for object of type `signed __fixed`

  `FIXED_BIT 16`

- minimum value for an object of type `signed __fixed`

  `FIXED_MIN (-0.5r-0.5r)`

- maximum value for an object of type `signed __fixed`

  `FIXED_MAX 0.999969482421875r`

- the difference between 0.0r and the least value greater than 0.0r that is representable in the `signed __fixed` type

  `FIXED_EPSILON 0.000030517578125r`

- maximum value for an object of type `unsigned __fixed`

  `UFIXED_MAX 0.999969482421875ur`

- the difference between 0.0r and the least value greater than 0.0r that is representable in the `unsigned __fixed` type

  `UFIXED_EPSILON 0.000030517578125ur`

- number of bits for object of type `signed long __fixed`

  `LFIXED_BIT 16`

- minimum value for an object of type `signed long __fixed`

  `LFIXED_MIN (-0.5R-0.5R)`

- maximum value for an object of type `signed long __fixed`

  `LFIXED_MAX 0.999969482421875R`

- the difference between 0.0R and the least value greater than 0.0R that is representable in the `signed long __fixed` type

  `LFIXED_EPSILON 0.000030517578125R`

- maximum value for an object of type `unsigned long __fixed`

  `ULFIXED_MAX 0.999969482421875UR`

- the difference between 0.0R and the least value greater than 0.0R that is representable in the `unsigned long __fixed` type

  `ULFIXED_EPSILON 0.000030517578125UR`

- number of bits for object of type `signed short __accum`

  `SACCUM_BIT 12`

- minimum value for an object of type `signed short __accum`

  `SACCUM_MIN (-8.0a-8.0a)`

- maximum value for an object of type `signed short __accum`

  `SACCUM_MAX 15.9921875a`

- the difference between 0.0a and the least value greater than 0.0a that is representable in the `signed short __accum` type

  `SACCUM_EPSILON 0.0078125a`

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

  `USACCUM_MAX 15.9921875ua`

- the difference between 0.0a and the least value greater than 0.0a that is representable in the `unsigned short __accum` type

  `USACCUM_EPSILON 0.0078125ua`

- number of bits for object of type `signed __accum`

  `ACCUM_BIT 20`

- minimum value for an object of type `signed __accum`

  `ACCUM_MIN (-8.0a-8.0a)`

- maximum value for an object of type `signed __accum`

  `ACCUM_MAX 15.999969482421875a`

- the difference between 0.0a and the least value greater than 0.0a that is representable in the `signed __accum` type

  `ACCUM_EPSILON 0.000030517578125a`

- maximum value for an object of type `unsigned __accum`

  `UACCUM_MAX 15.999969482421875ua`

- the difference between 0.0a and the least value greater than 0.0a that is representable in the `unsigned __accum` type

  `UACCUM_EPSILON 0.000030517578125ua`

- number of bits for object of type `signed long __accum`

  `LACCUM_BIT 20`

- minimum value for an object of type `signed long __accum`

  `LACCUM_MIN (-8.0A-8.0A)`

- maximum value for an object of type `signed long __accum`

  `LACCUM_MAX 15.999969482421875A`

- the difference between 0.0A and the least value greater than 0.0A that is representable in the `signed long __accum` type

  `LACCUM_EPSILON 0.000030517578125A`

- maximum value for an object of type `unsigned long __accum`

  `ULACCUM_MAX 15.999969482421875UA`

- the difference between 0.0A and the least value greater than 0.0A that is representable in the `unsigned long __accum` type

  `ULACCUM_EPSILON 0.000030517578125UA`

6 Language

6.1 Lexical elements

No additions in this section.

6.1.1 Keywords

Newly added *keywords*:

```
__accum  __fixed
__circ
__sat
```

In addition, target specific (implementation defined) memory space names should be added to this keywords list; as an example in this document we use the names `__X` and `__Y`.

6.1.2 Identifiers

No additions in this section.

6.1.2.1 Scopes of identifiers

No additions in this section.

6.1.2.2 Linkage of identifiers

No additions in this section.

6.1.2.3 Name spaces of identifiers

No additions in this section.

6.1.2.4 Storage durations of identifiers

No additions in this section.

6.1.2.5 Types

Additional types to the ISO C defined basic and arithmetic types are denoted as `short __fixed`, `__fixed`, `long __fixed`, `short __accum`, `__accum`, `long __accum`. Together these types will be named the fixedpoint types. For each of the `__fixed` and `__accum` types, there is a corresponding *signed* and *unsigned* type.

An object with `long __fixed` type is not necessarily capable of holding larger values than an object with `__fixed` type. In essence its scale will be larger, i.e. computations done in `long __fixed` arithmetic may produce identical or more precise results compared to computations done in `__fixed` arithmetic. For a definition of scale, see appendix A.

In the list of `short __fixed`, `__fixed`, `long __fixed`, the scale of each type is smaller than or equal to the scale of the next type in the list.

In the list of `short __accum`, `__accum`, `long __accum`, the integral part of each type shall not be larger than the integral part of the next type in the list. The scale of `short __accum` shall be equal to the scale defined for `short __fixed`. Likewise, the scale of `__accum` shall be equal to the scale of `__fixed`, and the scale of `long __accum` shall be equal to the scale of `long __fixed`.

For each of the *signed* types, there is a corresponding (but different) *unsigned* type that uses the same amount of storage (including sign information). For each *unsigned* type, the scale has the same size as its corresponding *signed* type, or one larger.

Types can be extended by addition of *memory-qualifiers.* Each existing type (including *const-qualified* and *volatile-qualified* types) can have a corresponding *memory-qualified* type for each existing memory qualifier. This creates a *memory-qualified* type, not a *qualified*-type.

General *memory-qualifiers* are defined, as an example we will call them `__X` and `__Y`, actual names are implementation defined. Especially `__X` and `__Y` are quite common in the world of DSP processors. A derived type is not qualified by the *memory-qualifiers* (if any) of the type from which it is derived (derived types are e.g. structures, unions and function return types).

To pointer types, an extra qualifier can be added, the `__circ`-qualifier, thus annotating the pointer to point to a circular array, with special address arithmetic behavior (this behavior is explained in Section 6.3.6).

The fixedpoint types can be extended with a saturation-qualifier `__sat`. This qualifier is only allowed with fixedpoint types `short __fixed`, `__fixed` and `long __fixed` and their unsigned versions. Saturation is further explained in Appendix A.3.

Example

The type designated as "`int *`" has type "pointer to `int`". The integer is found in an implementation defined memory. The `__X`-*memory-qualified* version of this type is designated as "`int * __X`" whereas the type designated as "`__X int *`" is not a *memory-qualified*-type — its type is "pointer to `__X`-*memory-qualified* `int`" and is a pointer to a *memory-qualified*-type.

The same holds for pointers to `__circ` arrays. The notation to create a pointer to a `__circ` array is:

```
__circ __X int * __Y p;
```

Meaning, 'p' is a pointer object, which pointer value is stored within `__Y`-memory. It is taken to point to an array within `__X`-memory, which is of `__circ int` type.

6.1.2.6 Compatible type and composite type

Additional rules for determining whether two types are compatible are described in 6.5.3.1 for memory-qualifiers, in 6.5.3.2 for saturation-qualifiers, and in 6.5.3.3 for circular-qualifiers.

6.1.3 Constants

Syntax

constant:
floating-constant
integer-constant
enumeration-constant
character-constant
fixed-constant

6.1.3.1 Floating constants

No additions in this section.

6.1.3.2 Integer constants

No additions in this section.

6.1.3.3 Enumeration constants

No additions in this section.

6.1.3.4 Character constants

No additions in this section.

6.1.3.5 Fixedpoint constants

Syntax

fixed-constant:
$\textit{digit-sequence}_{opt}$. *digit-sequence fixed-suffix*

digit-sequence:
digit
digit-sequence digit

fixed-suffix:
$\textit{unsigned-fixtype-suffix}_{opt}$ *fixtype-suffix*
$\textit{unsigned-fixtype-suffix}_{opt}$ *long-fixtype-suffix*

unsigned-fixtype-suffix: one of
u U

fixtype-suffix: one of

r a

longfixtype-suffix: one of
R A

All fixedpoint constants are of non-saturated type. To change the saturation-type, an explicit type cast should be used.

The type of a fixedpoint constant is the first of the corresponding list in which its value can be represented. Suffixed by the letter `r`: `__fixed`, `__accum`, `unsigned __accum`. Suffixed by the letter `a`: `__accum`, `unsigned __accum`. Suffixed by the letter `R`: `long __fixed`, `long __accum`, `unsigned long __accum`. Suffixed by the letter `A`: `long __accum`, `unsigned long __accum`. Suffixed by `ur` or `Ur`: `unsigned __fixed`, `unsigned __accum`. Suffixed by `ua` or `Ua`: `unsigned __accum`. Suffixed by `uR` or `UR`: `unsigned long __fixed`, `unsigned long __accum`. Suffixed by `uA` or `UA`: `unsigned long __accum`.

A `__fixed`-type value is in the range `[-1.0,+1.0>`.

Note: the unary minus is not part of the fixedpoint constant, therefore the notation `-1.0r` is not a valid `__fixed`-type constant, writing `(-0.5r-0.5r)` will fold to the desired value.

6.1.4 String literals

No additions in this section.

6.1.5 Operators

No additions in this section.

6.1.6 Punctuators

No additions in this section.

6.1.7 Header names

No additions in this section.

6.1.8 Preprocessing numbers

Description

Preprocessing numbers lexically include all floating, integer and fixedpoint constant tokens.

Semantics

A preprocessing number does not have type or a value. It acquires both after a successful conversion to a floating constant token, an integer constant token or a fixedpoint constant token.

6.1.9 Comments

No additions in this section.

6.2 Conversions

No additions in this section.

6.2.1 Arithmetic operands

When a `short __fixed` or `unsigned short __fixed` is used in an expression, it is first promoted to `__fixed` or `unsigned __fixed`, respectively. Likewise, when a `short __accum` or `unsigned short __accum` is used in an expression, it is first promoted to `__accum` or `unsigned __accum`, respectively. These conversions shall be value-preserving and thus do not affect the result of the expression.

6.2.1.1 Characters and integers

No additions in this section.

6.2.1.2 Signed and unsigned integers

No additions in this section.

6.2.1.3 Floating and integral

No additions in this section.

6.2.1.4 Floating types

No additions in this section.

6.2.1.5 Usual arithmetic conversions

In addition, for fixed type arithmetic conversions, see the Section 6.2.1.9.

No (un)usual conversions between integral and fixedpoint types are defined. Only explicit type conversions are defined.

6.2.1.6 Fixedpoint types

When a fixedpoint type is promoted to another fixedpoint type, if the value can be accurately represented by the new type, its value is unchanged.

When a fixedpoint type is promoted to another fixedpoint type, when the new type has a smaller scale, then the least significant bits of the value being converted are discarded to the size of the scale of the new type (for the definition of scale, see appendix A).

When a fixedpoint type is promoted to an unsigned fixedpoint type with equal or greater size, if the value being converted is non-negative, its value is unchanged. Otherwise, if the value being converted is negative, it is first converted to the signed fixedpoint type corresponding to the unsigned fixedpoint type and then converted to the unsigned fixedpoint type by adding or subtracting epsilon more than the maximum value that can be represented in the new type until the value is in the range of the new type.

When a fixedpoint type is promoted to a signed fixedpoint type with equal or greater size, if the value being converted is negative, its value is unchanged. Otherwise, if the value being converted can be represented by the new type, its value is unchanged. Otherwise, the result is implementation defined.

When a fixedpoint type is converted to a fixedpoint type with smaller size, if the value being converted can be represented by the smaller type (without looking at the precision of the two types), its value is unchanged. When the smaller type cannot represent the value, the result is implementation defined.

When needed (conversion from `__accum` to `__sat __fixed` types), saturation shall take place at the time of the conversion. These are the only conversions defined doing saturation on values. Saturation shall be done on the value which is converted into the new type, before doing the type conversion.

6.2.1.7 Fixedpoint and integral

Conversions from fixedpoint to integral types are value based.

During the conversion, the fractional part will be discarded. Since a `signed __fixed`-type object represents values in the range `[-1.0, +1.0>`, the only resulting integral values are `-1` or `0`. Since an `unsigned __fixed`-type object represents values in the range `[0, +1.0>`, the only resulting integral value is `0`.

If the integral part of the fixedpoint value cannot be represented by the integral type, the behavior is undefined.

When the value of an integral type is converted to fixedpoint, if the value being converted cannot be represented within the integral part of the fixedpoint, the result is undefined.

6.2.1.8 FixedPoint and floating

When a value with a fixedpoint type is converted to a floating point type, the result value is the nearest possible value representable by the new type.

When a value with floating point type is converted to fixedpoint, when the fixedpoint type can represent the original value (apart from precision), then the value is converted according to the specified floating point conversion rules.

When the floating point value is not representable in the fixedpoint type, the result is undefined. When converting the floating point value to a `__sat` FixedPoint type, no saturation is done.

This implies conversion from a value with floating type to a `__fixed`-type is always valid when the floating point value is within the range `[-1.0,+1.0>`. Any other values produce implementation defined results.

6.2.1.9 (Un)usual arithmetic conversions

Additional promotions are specified.

Constraints:

Automatic promotions between `(unsigned) long __fixed` and `(unsigned) __accum` types do not exist.

When all usual arithmetic conversions do not apply, then the following rules are taken into account:

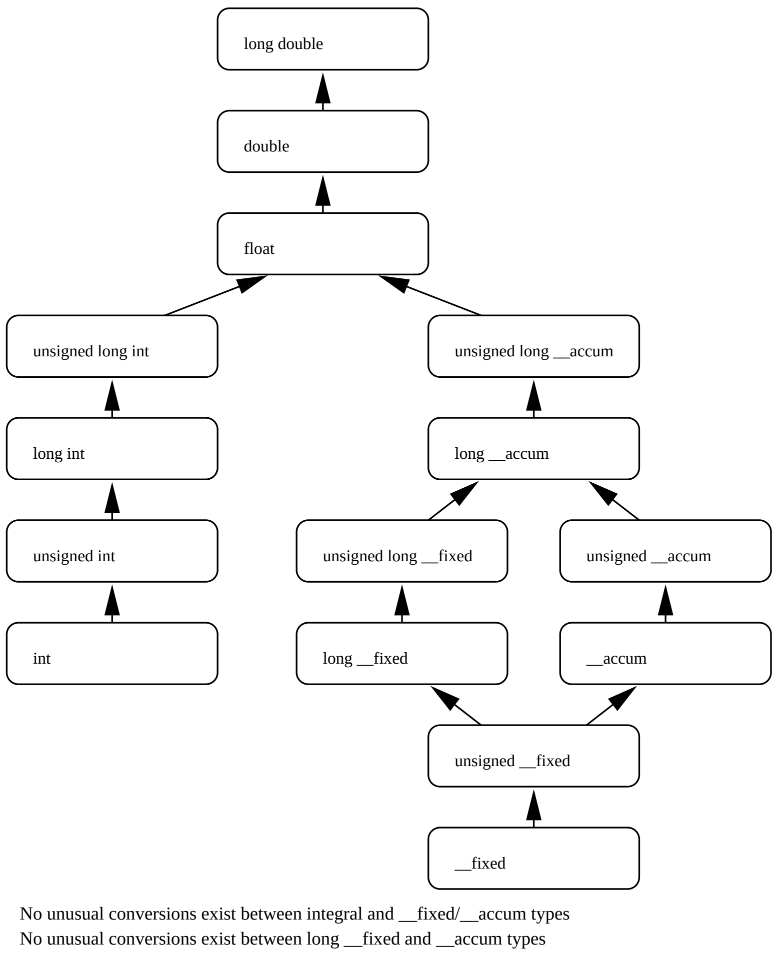

Figure 1: Unusual arithmetic conversions

If either operand has type `unsigned long __accum`, the other operand is converted to `unsigned long __accum`.

Otherwise, if one operand has type `long __accum` and the other has type `unsigned __accum`, if a `long __accum` can represent all values of an `unsigned __accum`, the operand of type unsigned __accum is converted to long __accum. If a long __accum cannot represent all values of an unsigned __accum, both operands are converted to unsigned long __accum.

Otherwise, if either operand has type long __accum, the other operand is converted to long __accum.

Otherwise, if either operand has type unsigned long __fixed, the other operand is converted to unsigned long __fixed.

Otherwise, if one operand has type long __fixed and the other has type unsigned __fixed, if a long __fixed can represent all values of an unsigned __fixed, the operand of type unsigned __fixed is converted to long __fixed. If a long __fixed cannot represent all values of an unsigned __fixed, both operands are converted to unsigned long __fixed.

Otherwise, if either operand has type unsigned __accum, the other operand is converted to unsigned __accum.

Otherwise, if one operand has type __accum and the other has type unsigned __fixed, if an __accum can represent all values of an unsigned __fixed, the operand of type unsigned __fixed is converted to __accum. If an __accum cannot represent all values of an unsigned __fixed, both operands are converted to unsigned __accum.

Otherwise, if either operand has type __accum, the other operand is converted to __accum.

Otherwise, if either operand has type unsigned __fixed, the other operand is converted to unsigned __fixed.

Otherwise, both operands have type __fixed.

The conversions are also shown in Figure 1. It should be read as:

If one operand of an operation has type of node 'x', and the other operand has a type of node 'y', then if node 'y' is in a subtree of node 'x', 'x' will be the type of the operation.

Otherwise, if node 'x' is in the subtree of node 'y', then 'y' will be the type of the operation.

Otherwise, no 'unusual' conversion is defined and the conversion is not a legal conversion.

6.2.1.10 Saturation promotions

Special promotions are specified, which only apply to the saturation-qualifier within an expression. The __sat qualifier is only valid on short __fixed, __fixed and long __fixed typed objects and expressions (signed and unsigned). The __sat qualifier does not apply to short __accum, __accum and long __accum typed objects and expressions.

Saturation on `signed __fixed` types saturate on the values `[-1.0,+1.0>`, while saturation on `unsigned __fixed` types saturate on the values `[0,+1.0>`.

The saturation qualified type of an expression result is inherited from its operands in the following way:

If either operand has `__sat` qualified type, and the expression has a `__fixed`-type result, then the expression result becomes `__sat` qualified.

Otherwise (both operands have non-`__sat` qualified type), then the expression result becomes non-`__sat` qualified.

During expression evaluation, saturation effects shall be effective before the result of the expression is used.

For assignment expressions, the result first is saturated according to the expression's type specification, then it is converted to the saturation type of the object assigned to. This to ensure the object always contains a valid value according to its *saturation-qualified* type.

6.2.2 Other operands

No additions in this section.

6.2.2.1 Lvalues and function designators

No additions in this section.

6.2.2.2 void

No additions in this section.

6.2.2.3 Pointers

For a description of the memory-qualifiers, see Section 6.5.3.1.

A pointer to non-memory-qualified `void` may be converted to or from a pointer to any incomplete or object type. A pointer to any incomplete or object type may be converted to a pointer to `void` and back again; the result shall compare equal to the original pointer.

When no memory-qualifier is defined for a pointer declaration, then the pointer shall be capable to address any object as defined by the normal ISO-C definition.

For any memory-qualifier m, whether a pointer to an m-qualified type may be converted to another memory-qualified type (with a different memory-qualifier) is implementation defined.

A memory-qualified pointer can be converted into an non-memory-qualified pointer to the same type. Conversion from an non-memory-qualified into the same memory-qualified pointer type is allowed.

A pointer value with *circular-qualified* type can be converted into (using a type cast) or assigned to a non-*circular-qualified* pointer type, the result will be the value of the original pointer without `__circ` behavior.

A pointer value with non-*circular-qualified* type can be converted to (using a type cast) or assigned to a *circular-qualified* pointer type. The *circular-qualified* result shall behave like it is a non-*circular-qualified* pointer value.

A pointer value to a *saturation-qualified* type may be converted into (using a type cast) or assigned to a to a non-*saturation-qualified* type, and vice versa.

Figure 2 shows when conversions are allowed, illegal or suspicious.

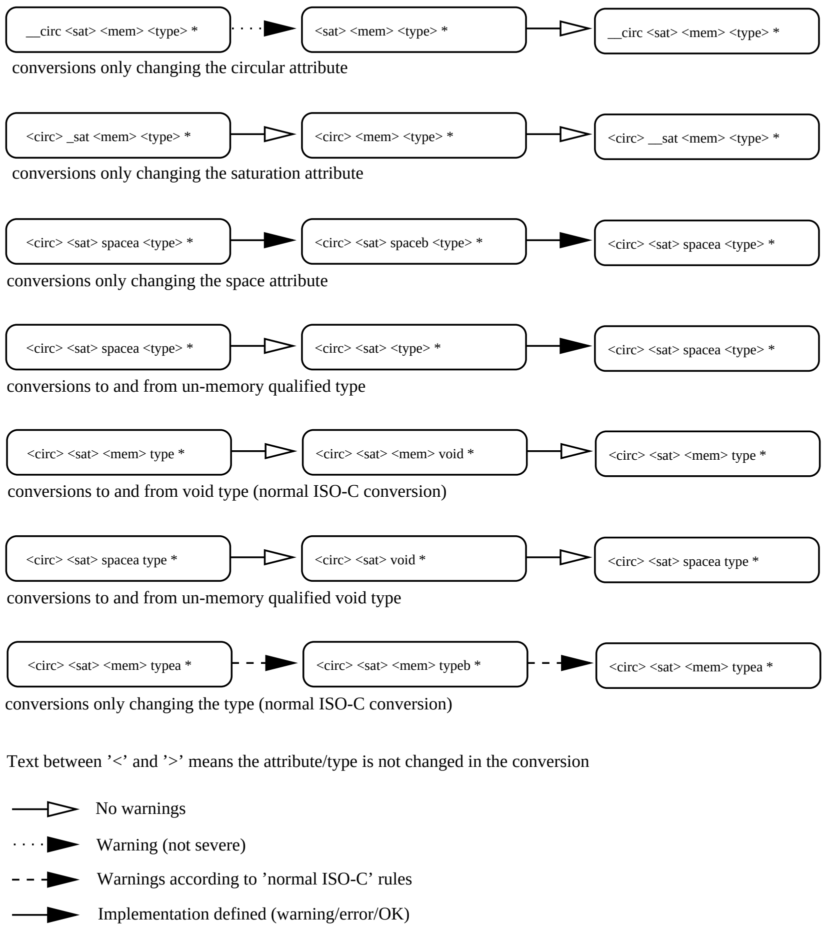

Figure 2: Allowed pointer type conversions

A `void *` declared object shall be capable of pointing to any type, except for `__circ` types (it will 'lose' the circular behavior). In this case a pointer declared as `__circ void *` shall be capable of pointing to all types of objects without restrictions.

6.3 Expressions

The unary operator `~` and the binary operators `<<`, `>>`, `&`, `^` and `|` are not allowed on fixedpoint expressions.

6.3.1 Primary expressions

No additions in this section.

6.3.2 Postfix operators

No additions in this section.

6.3.2.1 Array subscripting

In DSP-C, pointers and arrays may be declared using the `__circ` qualifier. The `__circ` qualifier is only allowed on arrays having one dimension. The semantics of subscripting in a circular array matches the ISO C definition in the sense that it is equivalent to circular pointer addition (see the ISO C definition). As a result, array subscripting expressions take care of an array being circular and will not address elements outside the array.

Example

```
int __circ x[5];
x[5] = 2;
```

will not effectively try to access the (non existing) element with index value 5. The element `x[0]` is assigned to. The same holds for pointer expressions, writing:

```
int __circ * p = x;
*(p+5) = 2;
```

will yield the same results as in the previous example.

6.3.2.2 Function calls

No default argument promotions are specified for fixedpoint types. For pointers, default argument promotion is to non-memory-qualified type.

6.3.2.3 Structure and union members

Structures and unions may have members with fixedpoint types. Circular arrays can be member of a structure or union. Pointers to circular arrays can be a member of a structure or union.

When a structure or union member has a *memory-qualified* type, this memory-qualifier does not affect the structure/union or its actual member. The structure or union itself can be defined having a memory-qualifier.

Example

```
struct {
    int __X value;      /* 'value' will reside within __Y memory,
                        * without issuing a warning
                        */
    int __X * p;        /* '*p' will actually fetch an object from
                        * __X memory (it points to __X), the
                        * pointer itself is stored in __Y memory
                        */
    } __Y str;
```

will entirely be allocated within `__Y` memory.

6.3.2.4 Postfix increment and decrement operators

No additions in this section.

6.3.3 Unary operators

No additions in this section.

6.3.3.1 Prefix increment and decrement operators

No additions in this section.

6.3.3.2 Address and indirection operators

The result of the `&` operator is a pointer to the object or function designated by its operand. The type of the pointer includes memory-qualifiers, circular and other qualifiers.

6.3.3.3 Unary arithmetic operators

The ~ can not be applied to fixedpoint values.

6.3.3.4 The `sizeof` operator

Pointers having different memory-qualifiers or different circular-qualifiers can have (and very likely have) different sizes. A pointer declared as `void * p` can have a size different from a pointer declared as `void __circ * p`, which can have a different size from a pointer declared as `void __circ __Y * p`. As such, the `sizeof` operator will often return different values for the different types of pointers.

6.3.4 Cast operators

Conversions between pointers with different memory-qualifiers may produce undefined results. Implicit conversions are not provided. Explicit conversions are accepted, but produce implementation defined results.

Conversion of a circular pointer to a non-circular pointer will lose its circular effect, even when later in the program the pointer is assigned to a circular pointer again. This implicit conversion will cause a compiler diagnostic.

Circular pointer objects are capable of containing non-circular pointer values, such a conversion is allowed. The circular pointer will then behave as if it is circular over the full possible address range, with an initial address value as assigned.

6.3.5 Multiplicative operators

No additions in this section.

6.3.6 Additive operators

For a circular pointer expression P, and an expression N of integral type, the following is additionally defined:

- the expressions N+(P) and (P)+N are equivalent
- if an expression P points to an element of a circular array, then the expression (P)+0 will point to that same element.
- if an expression P points to the last element of a circular array, then the expression (P)+1 will point to the first element of that array (and *not* to one past the last element as for ordinary arrays and pointers)
- if an expression P points to an element (not the last) of a circular array, then the expression (P)+1 points to the next element in that array
- if an expression P points to an element of a circular array, then the expression (P)+N (N's value is larger than 1 and not larger than the size of the circular array) points to the same element in that array as the expression ((P)+1)+(N-1)
- if an expression P points to the first element of a circular array, then the expression (P)-1 will point to the last element of that array
- if an expression P points to an element (not the first) of a circular array, then the expression (P)-1 points to the previous element in that array
- if an expression P points to an element of a circular array, then the expression (P)-N (N's value is larger than 1 and not larger than the size of the circular array) points to the same element in that array as the expression ((P)-1)-(N-1)

- when in one of the expressions (P)+N and (P)-N, N's value is larger than the size of the circular array, then it is undefined where that expression points to; the library functions _circ_add and _circ_sub are provided to handle expressions (P)+N and (P)-N in which N's value can be any value

Note that, as usual, when overflow occurs during the computation of an expression N above, the result is undefined.

6.3.7 Bitwise shift operators

Bitwise shifting is not allowed on fixedpoint values.

6.3.8 Relational operators

No additions in this section.

6.3.9 Equality operators

No additions in this section.

6.3.10 Bitwise AND operator

Bitwise operations are not allowed on fixedpoint expressions.

6.3.11 Bitwise exclusive OR operator

See comments in 6.3.10.

6.3.12 Bitwise inclusive OR operator

See comments in 6.3.10.

6.3.13 Logical AND operator

No additions in this section.

6.3.14 Logical OR operator

No additions in this section.

6.3.15 Conditional operator

In case of pointers, both operands should have compatible memory-qualifiers and circular-qualifiers.

6.3.16 Assignment operators

No additions in this section.

6.3.16.1 Simple assignment

No additions in this section.

6.3.16.2 Compound assignment

No additions in this section.

6.3.17 Comma operator

No additions in this section.

6.4 Constant expressions

No additions in this section.

6.5 Declarations

Syntax

declaration:
  declaration-specifiers init-declarator-list$_{opt}$;

declaration-specifiers:
  storage-class-specifier declaration-specifiers$_{opt}$
  type-specifier declaration-specifiers$_{opt}$
  type-qualifier declaration-specifiers$_{opt}$
  memory-qualifier declaration-specifiers$_{opt}$
  saturation-qualifier declaration-specifiers$_{opt}$
  circular-qualifier declaration-specifiers$_{opt}$

init-declarator-list:
  init-declarator
  init-declarator-list , *init-declarator*

init-declarator:
  declarator
  declarator = *initializer*

6.5.1 Storage-class specifiers

No additions in this section.

6.5.2 Type specifiers

Additional constraints

The set of type specifiers is extended with the following set:

- `short __fixed`, `signed short __fixed`, `unsigned short __fixed`
- `__fixed`, `signed __fixed`, `unsigned __fixed`
- `long __fixed`, `signed long __fixed`, `unsigned long __fixed`
- `short __accum`, `signed short __accum`, `unsigned short __accum`
- `__accum`, `signed __accum`, `unsigned __accum`
- `long __accum`, `signed long __accum`, `unsigned long __accum`

6.5.2.1 Structure and union specifiers

Syntax

specifier-qualifier-list:
  type-specifier specifier-qualifier-list$_{opt}$
  type-qualifier specifier-qualifier-list$_{opt}$
  saturation-qualifier specifier-qualifier-list$_{opt}$
  circular-qualifier specifier-qualifier-list$_{opt}$

6.5.2.2 Enumeration specifiers

No additions in this section.

6.5.2.3 Tags

No additions in this section.

6.5.3 Type qualifiers

No additions in this section.

6.5.3.1 Memory qualifiers

Syntax

memory-qualifier:
  `__X`
  `__Y`

The names `__X` and `__Y` should be replaced by implementation defined names. More names can exist. Within this document we use `__X` and `__Y` as two separated data memory spaces.

Constraints

At most one memory-qualifier shall appear in the same specifier list or qualifier list, either directly or via one or more `typedefs`.

Semantics

The properties associated with *memory-qualified* types are meaningful only for expressions that are lvalues.

The address of an object declared using one memory-qualifier can be assigned to a pointer declared as pointing to a type having the same memory-qualifier. The address of an object declared using a memory-qualifier can be assigned to a non-memory-qualified pointer to the same type. Whether the address of an object can be assigned to pointers having different memory-qualifiers is implementation defined.

6.5.3.2 Saturation qualifiers

Syntax

saturation-qualifier:
`__sat`

Constraints

The `__sat`-qualifier can only be specified for `__fixed`-type objects. At most one `__sat`-qualifier shall be specified for an object.

Semantics

The `__sat`-qualifier determines how arithmetic should be performed within arithmetic expressions. They do not affect the storage or representation of any object itself.

When no `__sat`-qualifier is specified, the default is non-`__sat`. Constants with a fixedpoint type are non-`__sat` qualified.

Saturation handling applies to all arithmetic operations resulting in a fixedpoint type.

6.5.3.3 Circular qualifiers

Syntax

circular-qualifier:
`__circ`

Constraints

The circular-qualifier shall not appear more than once in the same specifier list or qualifier list, either directly or via one or more `typedefs`. Only array types with one dimension and pointer types can be specified using this qualifier.

The circular-qualifier cannot be applied to:

- multi dimensional arrays
- simple type objects

Semantics

The circular-qualifier specifies that array subscripting or pointer addressing should perform modulo address arithmetic. This means that arithmetic on such an object can not run out of the array's boundaries. See Section 6.3.6 for the address arithmetic behavior of such pointer types.

6.5.4 Declarators

Syntax

declarator:
  pointer$_{opt}$ *direct-declarator*

direct-declarator:
  identifier
  (*declarator*)
  direct-declarator [*constant-expressions*$_{opt}$]
  direct-declarator (*parameter-type-list*)
  direct-declarator (*identifier-list*$_{opt}$)

pointer:
  * *type-qualifier-list*$_{opt}$
  * *type-qualifier-list*$_{opt}$ *pointer*

type-qualifier-list:
  type-qualifier
  memory-qualifier
  type-qualifier-list type-qualifier

parameter-type-list:
  parameter-list
  parameter-list , ...

parameter-list:
  parameter-declaration
  parameter-list , *parameter-declaration*

parameter-declaration:
  declaration-specifiers declarator
  declaration-specifiers abstract-declarator$_{opt}$

identifier-list:
  identifier
  identifier-list , *identifier*

6.5.4.1 Pointer declarators

See also Section 6.5.3.3.

6.5.4.2 Array declarators

See also Section 6.5.3.3.

6.5.4.3 Function declarators (including prototypes)

No additions in this section.

6.5.5 Type names

No additions in this section.

6.5.6 Type definitions

No additions in this section.

6.5.7 Initialization

No additions in this section.

6.6 Statements

No additions in this section.

6.6.1 Labeled statements

No additions in this section.

6.6.2 Compound statement, or block

No additions in this section.

6.6.3 Expression and null statements

No additions in this section.

6.6.4 Selection statements

No additions in this section.

6.6.4.1 The `if` statement

No additions in this section.

6.6.4.2 The `switch` statement

No additions in this section.

6.6.5 Iteration statements

No additions in this section.

6.6.5.1 The `while` statement

No additions in this section.

6.6.5.2 The `do` statement

No additions in this section.

6.6.5.3 The `for` statement

No additions in this section.

6.6.6 Jump statements

No additions in this section.

6.6.6.1 The `goto` statement

No additions in this section.

6.6.6.2 The `continue` statement

No additions in this section.

6.6.6.3 The `break` statement

No additions in this section.

6.6.6.4 The `return` statement

No additions in this section.

6.7 External definitions

No additions in this section.

6.7.1 Function definitions

No additions in this section.

6.7.2 External object definitions

Memory-qualifiers, saturation-qualifiers and other qualifiers used in external declarations should exactly match the qualifiers used in the object definition, otherwise the behavior is undefined.

6.8 Preprocessing directives

No additions in this section.

6.8.1 Conditional inclusion

No additions in this section.

6.8.2 Source file inclusion

No additions in this section.

6.8.3 Macro replacement

No additions in this section.

6.8.3.1 Argument substitution

No additions in this section.

6.8.3.2 The # operator

No additions in this section.

6.8.3.3 The ## operator

No additions in this section.

6.8.3.4 Rescanning and further replacement

No additions in this section.

6.8.3.5 Scope of macro definitions

No additions in this section.

6.8.4 Line control

No additions in this section.

6.8.5 Error directive

No additions in this section.

6.8.6 Pragma directive

No additions in this section.

6.8.7 Null directive

No additions in this section.

6.8.8 Predefined macro names

No additions in this section.

6.9 Future language directions

No additions in this section.

6.9.1 External names

No additions in this section.

6.9.2 Character escape sequences

No additions in this section.

6.9.3 Storage-class specifiers

No additions in this section.

6.9.4 Function declarators

No additions in this section.

6.9.5 Function definitions

No additions in this section.

6.9.6 Array parameters

No additions in this section.

6.10 Future language extensions

The following sections describe extensions which might be added in the future. Investigation is needed to determine if and how these should be added to DSP-C.

6.10.1 Positional storage qualifiers

Extra type qualifiers may be added, such as `__intern` and `__extern`, so the programmer can have more influence on the access time needed to access an object.

6.10.2 Register storage qualifiers

Extra storage qualifiers may be added, to force objects into specific registers or register sets.

6.10.3 Storage on known address

In computer systems, memory mapped devices are present. These can be addressed on fixed positions. Within standard C, there is no facility to directly address and name such hardware. Normally, this is done by creating a pointer object and initialize the pointer with a fixed value. This however does not prevent a linker to allocate another object on the specified address.

6.10.4 Complex types

The coming C9X standard introduces new types `complex`. Logically, DSP-C will be extended with new `complex` types, with fixedpoint real and imaginary values.

6.10.5 fixedpoint types

The representation and capabilities of `unsigned __fixed` types may be reconsidered.

Promotion rules for unsigned fixedpoint types where the scale is one larger than the scale of a signed fixedpoint type are hard to understand, and do introduce a change in value when converting from unsigned type into signed type. The conversion in itself needs a shift operation, thus causing the conversion to be relatively expensive as well. Possibly allowance of the unsigned fixedpoint types with this larger scale can be discarded.

Allowing `unsigned __fixed` types to represent values in the range `[0,+2.0>`. Not allowing this, causes one bit of storage not to be used (and must be ignored). However, allowing this range needs a definition of saturation effects (on `+1.0` or `+2.0`), and can cause algorithms not to be portable, when they use the effect that intermediate results can be `> +1.0`. Even so, adding the type can be reconsidered.

7 Library

7.1 Introduction

Since pointers can point to different memories within DSP-C, all standard functions expecting or returning a pointer as argument or return value will expect or return a pointer to an non-memory-qualified type.

All functions can be implemented according to the original ISO C definitions (normal non-memory-qualified pointers in their prototypes).

7.1.1 Definitions of terms

No additions in this section.

7.1.2 Standard headers

No additions in this section or subsections.

7.1.3 Errors `<errno.h>`

No additions in this section.

7.1.4 Limits `<float.h>` and `<limits.h>`

An extra header file `<fixed.h>` defines several macros that expand to various limits and parameters concerning fixedpoint types. The macros, their meanings, and the constraints on their values are listed in Section 5.2.4.3.

7.1.5 Common definitions `<stddef.h>`

No additions in this section.

7.1.6 Use of library functions

No additions in this section.

7.2 Diagnostics `<assert.h>`

No additions in this section or subsections.

7.3 Character handling `<ctype.h>`

No additions in this section or subsections.

7.4 Localization `<locale.h>`

No additions in this section or subsections.

7.5 Mathematics <math.h>

No additions in this section or subsections.

7.6 Non-local jumps <setjmp.h>

No additions in this section or subsections.

7.7 Signal handling <signal.h>

No additions in this section or subsections.

7.8 Variable arguments <stdarg.h>

No additions in this section or subsections.

7.9 Input/Output <stdio.h>

The `printf()` and `scanf()` formatters need to be extended with options to print `__fixed` and `__accum` types, as well as with special pointer types.

Defined is an addition of conversion specifiers with:

%hr	print/scan a `short __fixed` value
%r	print/scan a `__fixed` value
%lr	print/scan a `long __fixed` value
%ha	print/scan an `short __accum` value
%a	print/scan an `__accum` value
%la	print/scan a `long __accum` value
%hR	print/scan an `unsigned short __fixed` value
%R	print/scan an `unsigned __fixed` value
%lR	print/scan an `unsigned long __fixed` value
%hA	print/scan an `unsigned short __accum` value
%A	print/scan an `unsigned __accum` value
%lA	print/scan an `unsigned long __accum` value
%P	print/scan a `__circ` pointer value

For all fixedpoint types, output always contains a decimal-point. Width specifiers can be specified to specify the numbers of digits before the decimal-point and after the decimal-point, similar to the '%f' conversion specifier.

For `__circ` pointer values, the representation of the pointer value is implementation defined.

In case pointer types are not promoted to one common pointer type, an implementation may define more conversion specifiers to print/scan specific pointer types. In case all pointer types are promoted to one common pointer type, the address argument corresponding to the scanning value must be the address of such a common pointer type object.

7.10 General utilities <stdlib.h>

Extensions are defined, to add functions:

```
long __fixed atolfixed( const char * nptr );

long __accum atolaccum( const char * nptr );

long __fixed strtolfixed( const char * nptr, char **endptr );

long __accum strtolaccum( const char * nptr, char **endptr );

unsigned long __fixed atoulfixed( const char * nptr );

unsigned long __accum atoulaccum( const char * nptr );

unsigned long __fixed strtoulfixed( const char * nptr, char **endptr );

unsigned long __accum strtoulaccum( const char * nptr, char **endptr );
```

The set of `malloc()` functions can be extended to handle memory in each of the implementation defined memory spaces. Whether a complete set is offered is implementation defined. An implementation should at least provide a `malloc()` function which can allocate memory within a default memory (i.e. the memory can be addressed by using pointers declared as pointing to default memory).

7.11 String handling <string.h>

Using the non-memory-qualified pointer routines is not always optimal in execution speed. Therefore it is best to deliver a subset of the mostly used functions directly, in versions optimized for their specific use. Which specific routines are delivered is implementation defined.

Many implementations will want to define extra `memcpy()` functions for the various (combinations of) memory spaces, e.g.:

```
void __X * memcpy__X__X( void __X * s1, const void __X * s2, size_t n );

void __X * memcpy__X__Y( void __X * s1, const void __Y * s2, size_t n );

void __Y * memcpy__Y__X( void __Y * s1, const void __X * s2, size_t n );

void __Y * memcpy__Y__Y( void __Y * s1, const void __Y * s2, size_t n );
```

7.12 Date and time <time.h>

No additions in this section or subsections.

7.13 Fixedpoint support

An implementation can define fixedpoint support functions (e.g. for bitwise conversions to and from integral types), which the implementation will usually want to recognize as a compiler known function in order to generate efficient inline code for them.

Other features specific to a certain implementation can be defined in a similar manner, such as functions to use hardware-supported bitreverse or filter algorithms.

A Fixedpoint

This appendix describes how a fixedpoint value is defined and what it means to an implementation.

In principle there are four fixedpoint types:

- `signed __fixed`
- `unsigned __fixed`
- `signed __accum`
- `unsigned __accum`

A.1 `__fixed` types

The `signed __fixed` and `unsigned __fixed` types contain a mantissa value (value after the decimal point). The number of bits to represent this mantissa value is called the scale of the value. All fixedpoint values are stored in two's complement.

A `__fixed` object represent values in the range `[-1.0,+1.0>`. No special values (like the floating point NaN or Inf) are defined.

An `unsigned __fixed` object represents values in the range `[0.0,+1.0>`.

The `signed __accum` and `unsigned __accum` types are extensions to the types `signed __fixed` and respectively `unsigned __fixed`. They have equivalent behavior to the `__fixed` types, except they also have an integral part.

A.1.1 Representation

A fixedpoint type is completely characterized by three parameters:

Signedness Whether the type is signed or unsigned.

Size The total number of significant bits in the type. Note that this size can differ from the storage size of the type.

Scale The number of fractional bits in the type.

We denote the scale with s, the size with n and the bits as b_i. b_0 is the least significant bit, b_{n-1} is the most signicant bit.

The value of an unsigned fixedpoint value is given by

$$Value = 2^{-s} \sum_{i=0}^{n-1} 2^i b_i$$

The value of a signed fixedpoint value is given by

$$Value = 2^{-s}(-2^{n-1} b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i)$$

The value 2^{-s} corresponds to the type's `EPSILON` parameter from `<fixed.h>`.

The unsigned fixedpoint types shall have a scale equal to or one larger than the scale of the corrsponding signed type. If the scale is the same, the conversions between corresponding signed and unsigned types will not change representation.

If the scale is one larger, these conversions will need a shift of one bit to be value preserving.

Since unsigned fixedpoint types have an upper bound of 1.0, the version with the same scale has one unused bit in the representation. Whether this extra bit is interpreted as an integral part for unsaturated fixedpoint types is undefined.

A signed fixedpoint type needs one bit for the sign, therefore the scale can never be larger than the size minus one.

A.1.2 `signed __fixed` type

The `signed __fixed` types exist in three flavors:

Type	minimum size (in bits)	minimum scale (in bits)
`short __fixed`	8	7
`__fixed`	16	15
`long __fixed`	16	15

A `short __fixed` may be used in an expression wherever a `__fixed` may be used. The value is converted to a `__fixed`.

On `__fixed`-type objects, the *saturation-qualified* versions imply saturation to occur on the values -1.0 and (almost) +1.0.

A.1.3 `unsigned __fixed` type

An `unsigned __fixed` shall have the same scale as a `__fixed` or one larger. Since the unsigned version does not need a sign bit, the scale can be equal to the size.

The `signed __fixed` and `unsigned __fixed` types shall have same storage size. Because of the two possible implementations, the minimum scaling is as defined in the next table:

Type	minimum size (in bits)	minimum scale (in bits)
`unsigned short __fixed`	8	7 or 8
`unsigned __fixed`	16	15 or 16
`unsigned long __fixed`	16	15 or 16

An implementation should choose one of the possibilities, and apply this on all `__fixed` types.

An `unsigned short __fixed` may be used in an expression wherever an `unsigned __fixed` may be used. The value is converted to an `unsigned __fixed`.

For both `unsigned __fixed` implementations, the *saturation-qualified* versions imply saturation to occur on the values 0.0 and (almost) +1.0.

A.2 `__accum` types

A.2.1 `signed __accum` type

An `__accum` value is a `__fixed` value, extended with an integral part. The `__accum`-types shall have the same scaling factors as the corresponding `__fixed`-types. With an extension of 8 bits, an `__accum` value can represent values between `[-256.0,+256.0>`.

`signed __accum` types exist in three flavors:

Type	minimum size (in bits)	minimum scale (in bits)
`short __accum`	12	7
`__accum`	20	15
`long __accum`	20	15

The integral part of any `__accum` type shall not be less than 4 bits.

A `short __accum` may be used in an expression wherever an `__accum` may be used. The value is converted to an `__accum`.

A.2.2 `unsigned __accum` type

An `unsigned __accum` shall have the same scale as an `__accum` or one larger. Its scale shall be equal to the chosen `unsigned __fixed` implementation.

The `signed __accum` and `unsigned __accum` types should have the same storage size. Because of the two possible implementations, the minimum scaling is as defined in the next table:

Type	minimum size (in bits)	minimum scale (in bits)
`unsigned short __accum`	12	7 or 8
`unsigned __accum`	20	15 or 16
`unsigned long __accum`	20	15 or 16

The integral part of any `unsigned __accum` type shall not be less than 4 bits.

An `unsigned short __accum` may be used in an expression wherever an `unsigned __accum` may be used. The value is converted to an `unsigned __accum`.

A.3 Saturation

`__fixed` objects can be declared using the `__sat`-qualifier. This qualifier merely has its effect during computational actions done with such an object.

Saturation is only done when one or more operands of an operator are *saturation-qualified*, while the operation is done in a `__fixed` type.

Saturation on `signed __fixed` types will saturate to the values `[-1.0,+1.0>`. This means, when due to a computation, the result is larger than the upper bound value of a `signed __fixed` type, the result will be (almost) 1.0. When, due to a computation, the result is smaller than the lower bound of a `signed __fixed` value, the result will be -1.0.

Example

```
__sat signed __fixed a;
__sat signed __fixed b;
__sat signed __fixed c;
a = -0.75r;
b = -0.75r;
c = a + b;
/* c = -1.0r !!!  */
```

Saturation on `unsigned __fixed` types will saturate to the values `[0.0,+1.0>`. This means, when due to a computation, the result is larger than the upper bound value of an `unsigned __fixed` type, the result will be (almost) 1.0. When, due to a computation, the result is smaller than the lower bound of an `unsigned __fixed` value, the result will be 0.0.

Example

```
__sat unsigned __fixed a;
__sat unsigned __fixed b;
__sat unsigned __fixed c;
a = 0.50r;
b = 0.75r;
c = a - b;
/* c = 0.0r !!!  */
```

A.4 The file `<fixed.h>`

A new file **`<fixed.h>`** is defined with the following contents (the values are examples only and should be replaced by the proper values for an implementation):

/* Signed Fixed types */

/* `short __fixed` */

SFIXED_BIT	8	
SFIXED_MIN	(-0.5r-0.5r)	/* -1.0 */
SFIXED_MAX	0.9921875r	
SFIXED_EPSILON	0.0078125r	

/* `__fixed` */

FIXED_BIT	16	
FIXED_MIN	(-0.5r-0.5r)	/* -1.0 */
FIXED_MAX	0.999969482421875r	
FIXED_EPSILON	0.000030517578125r	

```
/* long __fixed */
LFIXED_BIT        32
LFIXED_MIN        (-0.5R-0.5R)                       /* -1.0 */
LFIXED_MAX        0.9999999995343387126922607421875R
LFIXED_EPSILON    0.0000000004656612873077392578125R
/* Unsigned Fixed types */
/* unsigned short __fixed */
USFIXED_MAX             0.9921875ur
USFIXED_EPSILON         0.0078125ur
/* unsigned __fixed */
UFIXED_MAX          0.999969482421875ur
UFIXED_EPSILON      0.000030517578125ur
/* unsigned long __fixed */
ULFIXED_MAX             0.9999999995343387126922607421875UR
ULFIXED_EPSILON         0.0000000004656612873077392578125UR
/* Signed Accum types */
/* short __accum */
SACCUM_BIT          16
SACCUM_MIN          (-128.0a-128.0a)    /* -256.0 */
SACCUM_MAX          255.9921875a
SACCUM_EPSILON      0.0078125a
/* __accum */
ACCUM_BIT         24
ACCUM_MIN         (-128.0a-128.0a)          /* -256.0 */
ACCUM_MAX         255.999969482421875a
ACCUM_EPSILON     0.000030517578125a
/* long __accum */
LACCUM_BIT        40
LACCUM_MIN        (-128.0A-128.0A)                         /* -256.0 */
LACCUM_MAX        255.9999999995343387126922607421875A
LACCUM_EPSILON    0.0000000004656612873077392578125A
/* Unsigned Accum types */
/* unsigned short __accum */
USACCUM_MAX             511.9921875ua
USACCUM_EPSILON         0.0078125ua
/* unsigned __accum */
UACCUM_MAX          511.999969482421875ua
UACCUM_EPSILON      0.000030517578125ua
/* unsigned long __accum */
ULACCUM_MAX             511.9999999995343387126922607421875UA
ULACCUM_EPSILON         0.0000000004656612873077392578125UA
```