

The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones

derek@knosof.co.uk

5.2.4.2.1 Sizes of integer types <limits.h>

integer types
sizes

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

Commentary

macro
object-like

In other words they represent object-like macros. The difference between this statement and one requiring that the values be integral constant expressions is that the `sizeof` and cast operators cannot appear in a `#if` preprocessing directive (or to be more exact, a sequence of preprocessing tokens in this context is not converted to tokens and `sizeof` or type names are not treated as such).

These macros were created to give names to values that developers are invariably aware of, may make use of, and for which they may have defined their own macros had the standard not provided any.

C++

17.4.4.2p2

All object-like macros defined by the Standard C library and described in this clause as expanding to integral constant expressions are also suitable for use in `#if` preprocessing directives, unless explicitly stated otherwise.

18.2.2p2

Header <limits> (Table 16): . . . The contents are the same as the Standard C library header <limits.h>

Other Languages

The equivalent identifiers in Java contain constant values. Ada and Fortran (90) use a function notation for returning representation properties of types and objects.

Common Implementations

These values can all be implemented using strictly conforming C constructs (although the numerical values used may vary between implementations). That is not to say that all vendors' implementations do it in this way. For some of these macros it is possible to use the same definition in all environments (without affecting the existing conformance status of programs). However, this usage is not very informative:

```
1 #define UINT_MAX (-1U)
2 #define ULONG_MAX (-1UL)
3 #define ULLONG_MAX (-1ULL)
```

The definition of the macros for the signed types always depends on implementation-defined characteristics. For instance,

```
1 #define INT_MAX 32767 /* 16 bits */
2 #define INT_MIN (-32767-1)
```

Coding Guidelines

The standard specifies values, not the form the expression returning those values takes. For instance, the replacement for `INT_MIN` is not always the sequence of digits denoting the actual value returned.

int³¹⁷
minimum value

Table 303.1: Number of identifiers defined as macros in <limits.h> (see Table ?? for information on the number of identifiers appearing in the source) appearing in the visible form of the .c and .h files.

Name	.c file	.h file	Name	.c file	.h file	Name	.c file	.h file
LONG_MAX	47	28	CHAR_MAX	15	8	CHAR_BIT	36	3
INT_MAX	106	17	INT_MIN	17	7	SCHAR_MIN	12	2
UINT_MAX	30	14	UCHAR_MAX	16	5	LLONG_MAX	0	1
SHRT_MAX	20	13	CHAR_MIN	9	5	ULLONG_MAX	0	0
SHRT_MIN	19	12	SCHAR_MAX	13	4	LLONG_MIN	0	0
USHRT_MAX	12	11	MB_LEN_MAX	15	4			
ULONG_MAX	85	10	LONG_MIN	23	3			

- 304 Moreover, except for **CHAR_BIT** and **MB_LEN_MAX**, 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 integer promotions.

*_MAX
same type as
*_MIN
same type as

Commentary

CHAR_BIT and MB_LEN_MAX have no implied integer type. This requirement maintains the implicit assumption that use of one of these identifiers should not cause any surprises to a developer. There could be some surprises if promotions occurred; for instance, if the constants had type **unsigned long long**. There is no literal suffix to indicate a character or **short** type, so the type can only be the promoted type.

Common Implementations

Suffixes are generally used, rather than hexadecimal notation, to specify unsigned types.

- 305 Their implementation-defined values shall be equal or greater in magnitude (absolute value) to those shown, with the same sign.

Commentary

The C Standard does not specify the number of bits in a type. It specifies the minimum and maximum values that an object of that type can represent. An implementation is at liberty to exceed the limits specified here. It cannot fail to meet them. Except for the character types, a type may also contain more bits in its object representation than in its value representation.

integer types
relative ranges

object repre-
sentation
value repre-
sentation

Common Implementations

The values that are most often greater than the ones shown next are those that apply to the type **int**. On hosted implementations they are often the same as the corresponding values for the type **long**. On a freestanding implementation the processors' efficiency issues usually dictate the use of smaller numeric ranges, so the minimum values shown here are usually used. The values used for the corresponding character, **short**, **long**, and **long long** types are usually the same as the ones given in the standard.

The Unisys A Series^[5] is unusual in not only using sign magnitude, but having a single size (six bytes) for all non-character integer types (the type **long long** is not yet supported by this vendor's implementation).

```

1 #define SHRT_MIN      (-549755813887)
2 #define SHRT_MAX      549755813887
3 #define USHRT_MAX     549755813887U
4 #define INT_MIN       (-549755813887)
5 #define INT_MAX       549755813887
6 #define UINT_MAX      549755813887U
7 #define LONG_MIN      (-549755813887L)
8 #define LONG_MAX      549755813887L
9 #define ULONG_MAX     549755813887UL

```

The character type use two's complement notation and occupies a single byte.

The C compiler for the Unisys e-@ction Application Development Solutions (formerly known as the Universal Compiling System, UCS)^[6] has 9-bit character types— 18-bit **short**, 36-bit **int** and **long**, and 72-bit **long long**.

Coding Guidelines

Some applications may require that implementations support a greater range of values than the minimum requirements specified by the standard. It is the developer's responsibility to select an implementation that meets the application requirements in this area.

These, and other, macro names defined by the standard provide symbolic names for the quantities they represent. The contexts in which these identifiers (usually macros) might be expected to occur (e.g., as operands of certain operators) are discussed in subsections associated with the C sentence that specifies them.

Example

Whether an occurrence of a symbolic name is making use of representation information is not always clear-cut.

```

1  #include <limits.h>
2
3  extern _Bool overflow;
4  extern int total;
5
6  void f(int next_digit_val)
7  {
8  /*
9   * Check that the calculation will probably not overflow. Perform
10  * arithmetic on INT_MAX, but use a property (being a factor of
11  * ten-less-than), not any specific numeric value. The symbolic
12  * name is still being treated as the maximum representable int value.
13  */
14  if (total < (INT_MAX / 10))
15      total = (total * 10) + next_digit_val;
16  else
17      overflow = 1;
18  }
19
20  unsigned int g(unsigned long some_val)
21  {
22  /*
23   * While the internal representation of the value of UINT_MAX
24   * is required to have all bits set, this might be considered
25   * incidental to the property it is defined to represent.
26   */
27  return (unsigned int)(some_val & UINT_MAX);
28  }
29
30  void h(int *characteristic)
31  {
32  /*
33   * Comparing symbolic names against individual literal values is often
34   * an indication that use is being made of representation details. If
35   * the comparisons occur together, as a set, it might be claimed that an
36   * alternative representation of the characteristics of an implementation
37   * are being deduced. The usage below falls into the trap of overlooking
38   * an undefined behavior (unless a translator has certain properties).
39   */
40  switch (INT_MIN)
41  {
42      case -32767: *characteristic=1;
43                  break;
44
45      case -32768: *characteristic=2;
```

symbolic
name

```

46             break;
47
48     case -2147483647: *characteristic=3;
49             break;
50
51     default: *characteristic=4;
52             break;
53     }
54 }
```

306 14) See “future language directions” (6.11.3).

footnote
14

307 — number of bits for smallest object that is not a bit-field (byte)

CHAR_BIT
macro

CHAR_BIT 8

Commentary

C is not wedded to an 8-bit byte, although this value is implicit in a large percentage of source written in it.

Common Implementations

On most implementations a byte occupies 8 bits. The POSIX Standard requires that CHAR_BIT have a value of 8. The Digital DEC 10 and Honeywell/Multics^[1] used a 36-bit word with the underlying storage organization based on 9-bit bytes. Some DSP chips have a 16- or 32-bit character type (this often has more to do with addressability issues than character set sizes).

Coding Guidelines

A value of 8, for this quantity, is welded into many developers’ mind-set. Is a guideline worthwhile, just to handle those cases where it does not hold? Developers porting programs from an environment where CHAR_BIT is 8, to an environment where it is not 8 are likely to uncover many algorithmic dependencies. In many application domains the cost of ensuring that programs don’t have any design, or algorithmic, dependencies on the number of bits in a byte is likely to be less than the benefit (reduced costs should a port to such an environment be undertaken).

A literal appearing within source code conveys no information on what it represents. If the number of bits in a byte is part of a calculation, use of a name, which has been defined, by the implementation to represent that quantity (even if it is not expected that the program will ever be translated in an environment where the value differs from 8) provides immediate semantic information to the developer reading the source (saving the effort of deducing it).

The CHAR_BIT is both a numeric quantity and represents bit-level information.

Example

In the first function assume a byte contains 8 bits. The second function has replaced the literal that explicitly contains this information, but not the literal that implicitly contains it. The third function contains an implicit assumption about the undefined behavior that occurs if `sizeof(int) == sizeof(char)`.

```

1  #include <limits.h>
2
3  unsigned char second_byte_1(unsigned int valu)
4  {
5  return ((valu >> 8) & 0xff);
6  }
7
8  unsigned char second_byte_2(unsigned int valu)
9  {
10 return ((valu >> CHAR_BIT) & 0xff);
```

```

11 }
12
13 unsigned char second_byte_3(unsigned int valu)
14 {
15     return ((valu >> CHAR_BIT) & ((0x01 << CHAR_BIT) - 1));
16 }

```

SCHAR_MIN
value

— minimum value for an object of type **signed char**

308

SCHAR_MIN -127 // $-(2^7-1)$

Commentary

The minimum value an 8-bit, one's complement or sign magnitude representation can support.

Common Implementations

A value of -128 is invariably used in a two's complement representation.

Coding Guidelines

two's complement
minimum value

Because almost all implementations use two's complement, this value is often assumed to be -128. Porting to processors that use a different integer representation is likely to be very rare event. Designing code to handle both cases increases cost and complexity (leading to faults) for a future benefit that is unlikely to be cashed in. Ignoring the additional value available in two's complement implementations, on the grounds of symmetry or strict standard's conformance, can lead to complications if this value can ever be created (e.g., by a cast from a value having greater rank).

— maximum value for an object of type **signed char**

309

SCHAR_MAX +127 // 2^7-1

UCHAR_MAX

— maximum value for an object of type **unsigned char**

310

UCHAR_MAX 255 // 2^8-1

Commentary

Because there are two representations of zero in one's complement and signed magnitude the relation $(\text{SCHAR_MAX} - \text{SCHAR_MIN}) < \text{UCHAR_MAX}$ is true for all implementations targeting such environments.

In the case of two's complement the equality $(\text{SCHAR_MAX} - \text{SCHAR_MIN}) == \text{UCHAR_MAX}$ is true, provided there are no padding bits in the representation of the type **signed char** (padding bits are never permitted in the representation of the type **unsigned char**, and no implementations known to your author use padding bits in the type **signed char**).

There is a requirement that UCHAR_MAX equals $2^{\text{CHAR_BIT}} - 1$. Equivalent requirements do not hold for USHRT_MAX or UINT_MAX

A byte is commonly thought of as taking on the range of values between zero and UCHAR_MAX and occupying CHAR_BIT bits. In other words it is treated as an unsigned type.

Other Languages

The Java class `java.lang.Character` contains the member:

```

1 public static final char MAX_VALUE = '\uffff';

```

unsigned
char
pure binary

unsigned
char
pure binary

2^{316}
USHRT_MAX
 2^{319}
UINT_MAX
byte
addressable unit

Coding Guidelines

The macro `UCHAR_MAX` has a value that has an arithmetic property. It also has a bitwise property— all bits set to one. The property of all bits set to one is often seen in code that performs bit manipulation (the issue of replacing numeric literals with meaningful identifiers is dealt with elsewhere).

all bits one

symbolic
name

311— minimum value for an object of type `char`

char
minimum value
CHAR_MIN

`CHAR_MIN` *see below*

Commentary

The variability in the value of this macro follows the variability of the choice of representation for the type `char`. It depends on whether `char` is treated as a signed or unsigned type. There is an equivalent set of macros for wide characters.

³²⁶ `char`
if treated as
signed integer

Other Languages

`java.lang.Character` contains the member:

```
1 public static final char MIN_VALUE = '\u0000';
```

Common Implementations

The value used is often translation-time selectable, via a conditional inclusion in the <limits.h> header. The translator using a developer-accessible option to control the choice of type.

312— maximum value for an object of type `char`

CHAR_MAX

`CHAR_MAX` *see below*

Commentary

The commentary on `CHAR_MIN` is applicable here.

³¹¹ `char`
minimum value

Other Languages

`java.lang.Character` contains the member:

```
1 public static final char MAX_VALUE = '\uffff';
```

313— maximum number of bytes in a multibyte character, for any supported locale

MB_LEN_MAX

`MB_LEN_MAX` 1

Commentary

For the value of this macro to be a translation-time constant, it has to refer to the locales known to the translator. The value must be large enough to support the sequence of bytes needed to specify a change from any supported shift state, to any other supported shift state, for any character available in the later shift state. There is no requirement on the implementation to support redundant changes of shift state.

footnote
152

Common Implementations

Many US- and European-based vendors use a value of 1. The GNU library uses a value of 16.

Coding Guidelines

Using this identifier as the size in the definition, of an array used to hold the bytes of a multibyte character, implies that the bytes stored will not contain any redundant shift sequences.

short
minimum value— minimum value for an object of type **short int**

314

SHRT_MIN -32767 // $-(2^{15}-1)$ **Commentary**

The minimum value a 16-bit, one's complement or sign magnitude representation can support.

Other Languages

Java defines **short** as having the full range of a two's complement 16-bit value.

Common Implementations

Most implementations support a 16-bit **short** type.

The token `-` is a unary operator; it is not part of the integer constant token. The expression `-32768` has type **long** (assuming a 16-bit **int** type), `((-32767)-1)` has type **int** (assuming that two's complement notation is being used). A value of `(-32767-1)` is invariably used in a 16-bit two's complement representation.

— maximum value for an object of type **short int**

315

SHRT_MAX +32767 // $2^{15}-1$ **Commentary**short³¹⁴
minimum value

The commentary on SHRT_MIN is applicable here.

USHRT_MAX

— maximum value for an object of type **unsigned short int**

316

USHRT_MAX 65535 // $2^{16}-1$ **Commentary**object rep-
resentation

There is no requirement that USHRT_MAX equals $2^{\text{sizeof}(\text{short}) \times \text{CHAR_BIT}} - 1$.

Common Implementations

The standard does not prohibit an implementation giving USHRT_MAX the same value as SHRT_MAX (e.g., Unisys A Series^[5]). In practice the additional bit used to represent sign information, in a signed type, is invariably used to represent a greater range of positive values for unsigned types.

int
minimum value— minimum value for an object of type **int**

317

INT_MIN -32767 // $-(2^{15}-1)$ **Commentary**

The minimum value a 16-bit, one's complement or sign magnitude representation can support.

Other Languages

Java defines **int** as holding the full range of a 32-bit two's complement value. `java.lang.Integer` contains the member:

```
1 public static final int MIN_VALUE = 0x80000000;
```

Common Implementationsshort³¹⁴
minimum value

On most implementations this macro is the same as either SHRT_MIN or LONG_MIN. A value of `(-32767-1)` is invariably used in a 16-bit two's complement representation, while a value of `(-2147483647-1)` is invariably used in a 32-bit two's complement representation.

A number of DSP processors use 24 bits to represent the type **int**.^[3]

Coding Guidelines

Developers often make some fundamental assumptions about the representation of `int`. These assumptions may be true in the environment in which they spend most of their time developing code, but in general are always true. Coding to the minimum limit specified in the standard in a 32-bit environment can be very restrictive and costly. If a program is never going to be ported to a non-32-bit host the investment cost of writing code to the minimum requirements, guaranteed by the standard, may never reap a benefit. This is a decision that is outside the scope of these coding guidelines.

318— maximum value for an object of type `int`

INT_MAX

```
INT_MAX +32767 // 215-1
```

Other Languages

`java.lang.Integer` contains the member:

```
1 public static final int MAX_VALUE = 0x7fffffff;
```

Common Implementations

On most implementations this macro is the same as either `SHRT_MAX` or `LONG_MAX`.

319— maximum value for an object of type `unsigned int`

UINT_MAX

```
UINT_MAX 65535 // 216-1
```

Commentary

There is no requirement that `UINT_MAX` equal $2^{\text{sizeof(int)} \times \text{CHAR_BIT}} - 1$.

object representation

Common Implementations

The standard does not prohibit an implementation giving `UINT_MAX` the same value as `INT_MAX`, which would entail increasing the value of `INT_MAX` (e.g., Unisys A Series^[5]). In practice the bit used to represent sign information, in a signed type, is invariably included in the value representation for unsigned types (enabling a greater range of positive values to be represented). On most implementations this macro is the same as either `USHRT_MAX` or `ULONG_MAX`.

Coding Guidelines

The term *word* has commonly been used in the past to refer to a unit of storage that is larger than a byte. The unit of storage chosen is usually the *natural* size for a processor, much like the suggestion for the size of an `int` object. Like a byte, a word is usually considered to be unsigned, taking on the range of values between zero and `UINT_MAX`.

word
range of valuesint
natural size

Like the `UCHAR_MAX` macro the `UINT_MAX` macro is sometimes treated as having the bitwise property of all bits set to one.

310
UCHAR_MAX
310 all bits one

320— minimum value for an object of type `long int`

```
LONG_MIN -2147483647 // -(231-1)
```

Commentary

The minimum value a 32-bit, one's complement or sign magnitude representation can support.

Other Languages

`java.lang.Long` contains the member:

```
1 public static final long MIN_VALUE = 0x8000000000000000L;
```

Common Implementations

A value of $(-2147483647-1)$ is invariably used in a 32-bit two's complement representation. Prior to the introduction of the type **long long** in C99, some implementations targeting 64-bit processors chose to use the value $(-9223372036854775807-1)$, while others extended the language to support a **long long** type and kept long at 32 bits.

The number of bits used to represent the type **long** is not always the same as, or an integer multiple of, the number of bits in the type **int**. The ability to represent a greater range of values (than is possible in the type **int**) may be required, but processor costs may also be a consideration. For instance, the Texas Instruments TMS320C6000,^[4] a DSP processor, uses 32 bits to represent the type **int** and 40 bits to represent the type **long** (this choice is not uncommon). Those processors (usually DSP) that use 24 bits to represent the type **int**,^[3] often use 48 bits to represent the type **long**. The use of 24/48 bit integer type representations can be driven by application requirements where a 32/64-bit integer type representation are not cost effective.^[2]

Coding Guidelines

Some developers assume that `LONG_MIN` is the most negative value that a program can generate. The introduction of **long long**, in C99, means that this assumption is no longer true.

widest type
assumption

— maximum value for an object of type **long int**

321

`LONG_MAX +2147483647 // 231-1`

Other Languages

`java.lang.Long` contains the member:

```
    public static final long MAX_VALUE = 0x7fffffffffffffffL;
```

Common Implementations

The value given in the standard is the one almost always used by implementations.

— maximum value for an object of type **unsigned long int**

322

`ULONG_MAX 4294967295 // 232-1`

Commentary

There is no requirement that `ULONG_MAX` equal $2^{\text{sizeof}(\text{long}) \times \text{CHAR_BIT}} - 1$.

Common Implementations

The standard does not prohibit an implementation giving `ULONG_MAX` the same value as `LONG_MAX` (Unisys A Series^[5]). In practice the additional bit used to represent sign information, in a signed type, is invariably used to represent a greater range of positive values for unsigned types.

The value given in the standard is the one almost always used by implementations.

Coding Guidelines

Some developers assume that `ULONG_MAX` is the largest value that a program can generate. The introduction of the type **long long** means that this assumption is no longer true.

object rep-
resentation

widest type
assumption

— minimum value for an object of type **long long int**

323

`LLONG_MIN -9223372036854775807 // -(263-1)`

Commentary

The minimum value a 64-bit, one's complement or sign magnitude representation can support. This value is calculated in a way that is consistent with the other `_MIN` values. However, hardware support for 64-bit integer arithmetic is usually only available on modern hosts, all of which use two's complement (to the best of your author's knowledge).

C90

Support for the type **long long** and its associated macros is new in C99.

C++

The type **long long** is not available in C++ (although many implementations support it).

Other Languages

In Java the type **long** uses 64 bits; a **long long** type is not needed (unless support for 128-bit integers were added to the language).

Common Implementations

This macro was supported in those C90 implementations that had added support for **long long** as an extension.

Coding Guidelines

The **long long** data type is new in C99. It will be some time before it is widely supported. A source file that accesses this identifier will fail to translate with a C90 implementation. The issue encountered is likely to be one of nontranslation, not one of different behavior. Deciding to create programs that require at least a 64-bit data type is an application-based issue that is outside the scope of these coding guidelines.

324— maximum value for an object of type **long long int**

```
LLONG_MAX +9223372036854775807 // 263-1
```

C90

Support for the type **long long** and its associated macros is new in C99.

C++

The type **long long** is not available in C++ (although many implementations support it).

325— maximum value for an object of type **unsigned long long int**

```
ULLONG_MAX 18446744073709551615 // 264-1
```

Commentary

There is no requirement that `ULLONG_MAX` equals $2^{\text{sizeof}(\text{long long}) \times \text{CHAR_BIT}} - 1$.

object representation

C90

Support for the type **unsigned long long** and its associated macros is new in C99.

C++

The type **unsigned long long** is not available in C++.

326 If the value of an object of type **char** is treated as a signed integer when used in an expression, the value of `CHAR_MIN` shall be the same as that of `SCHAR_MIN` and the value of `CHAR_MAX` shall be the same as that of `SCHAR_MAX`.

char
if treated as
signed integer

Commentary

In most implementations the value of objects of type **char** will promote to the type **signed int**. So the values of objects of type **char** are treated as a **signed integer**. It is only when an object of type **char** is treated as an lvalue, that its signedness becomes a consideration. For instance, on being assigned to, is modulo arithmetic or undefined behavior used for out-of-range values.

integer promotions

lvalue

Common Implementations

Most implementations provide a translation-time option that enables a developer to select how the type **char** is to be handled. This option usually causes the translator to internally predefine an object-like macro. The existence of this macro definition is tested inside the <limits.h> header to select between the two possible values of the CHAR_MIN and CHAR_MAX macros.

Coding Guidelines

A comparison of the value of CHAR_MIN against SCHAR_MIN (or CHAR_MAX against SCHAR_MAX) can be used to determine an implementation-defined behavior (whether the type **char** uses the same representation as **signed char** or **unsigned char**).

Example

```

1  #include <limits.h>
2  #include <stdio.h>
3
4  int main(void)
5  {
6      if (CHAR_MIN == SCHAR_MIN)
7          {
8              printf("char appears to be signed\n");
9              if (CHAR_MAX != SCHAR_MAX)
10                 printf("nonconforming implementation\n");
11             }
12     if (CHAR_MIN != SCHAR_MIN)
13         printf("char does not appear to be signed\n");
14     }

```

Otherwise, the value of CHAR_MIN shall be 0 and the value of CHAR_MAX shall be the same as that of UCHAR_MAX.¹⁵⁾ 327

Commentary

When the type **char** supports the same range of values as the type **unsigned int**, the type of CHAR_MIN must also be **unsigned int** (i.e., the body of its definition is likely to be 0u).

* MIN³⁰⁴
same type as

Coding Guidelines

The possibility that the expression (CHAR_MIN == 0) && (CHAR_MIN <= -1) can be true is likely to be surprising to developers. However, implementations where this macro has type **unsigned int** are not sufficiently common to warrant a guideline recommendation (e.g., perhaps suggesting that CHAR_MIN always be cast to type **int**, or that it not appear as the operand of a relational operator).

Example

```

1  #include <limits.h>
2  #include <stdio.h>
3
4  int main(void)
5  {
6      if (CHAR_MIN == 0)
7          {
8              printf("char appears to be unsigned\n");
9              if ((CHAR_MAX - UCHAR_MAX) != 0)
10                 printf("nonconforming implementation\n");
11             }
12     }

```

```
13  if (((int)CHAR_MIN) <= -1) /* Handle the case #define CHAR_MIN 0U */
14      printf("char does not appear to be unsigned\n");
15  }
```

328 The value `UCHAR_MAX` shall equal $2^{\text{CHAR_BIT}} - 1$.

`UCHAR_MAX`
value

Commentary

Any unsigned character types are not allowed to have any hidden bits that are not used in the representation. Therefore `UCHAR_MAX` must equal $2^{\text{sizeof(char)} \times \text{CHAR_BIT}} - 1$. Since `sizeof(char) == 1` by definition, the above C specification must hold.

unsigned
char
pure binary

sizeof char
defined to be 1

C90

This requirement was not explicitly specified in the C90 Standard.

C++

Like C90, this requirement is not explicitly specified in the C++ Standard.

Other Languages

Most languages do not get involved in specifying this level of detail.

329 **Forward references:** representations of types (6.2.6), conditional inclusion (6.10.1).

References

1. H. Bull. *Multics C User's Guide*. Honeywell Bull, Inc, 1987.
2. J. A. Moorer. 48-bit integer processing beats 32-bit floating point for professional audio applications. In *AES 107th Convention*, Sept. 1999. preprint 5038.
3. Motorola, Inc. *DSP5600 24-bit Digital Signal Processor Family Manual*. Motorola, Inc, Austin, TX, USA, dsp56kfamum/ad edition, 1995.
4. Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*. Texas Instruments, spru189f edition, Oct. 2000.
5. Unisys Corporation. *C Programming Reference Manual, Volume 1: Basic Implementation*. Unisys Corporation, 8600 2268-203 edition, 1998.
6. Unisys Corporation. *C Compiler Programming Reference Manual Volume 1: C Language and Library*. Unisys Corporation, 7831 0422-006, release level 8R1A edition, 2001.