X3J11/95-002
WG14/N401

Document Number: WG14 N401/X3J11 95-002

C9X Revision Proposal

Title: Proposed <inttypes.h> header

Author: John W. Kwan

Author Affiliation: Hewlett-Packard Company

Postal Address: 11000 Wolfe Road
Cupertino, California 95014

E-mail Address: jkwan@cup.hp.com

Telephone Number: +1 408 4476442

Fax Number: +1 408 4474924

Sponsor: X3J11

Date: 15 March 1995

Proposal Category:
- __ Editorial change/non-normative contribution
- __ Correction
- X_ New feature
- __ Addition to obsolescent feature list
- __ Addition to Future Directions
- __ Other (please specify)

Area of Standard Affected:
- __ Environment
- __ Language
- __ Preprocessor
- __ Library
  - __ Macro/typedef/tag name
  - __ Function
  - X_ Header

Prior Art: This header is available, or will be available, in a number of implementations

Target Audience: Software developers writing portable programs

Related Documents : (none)

Proposal Attached: X_ Yes, but what's your interest?

Abstract:

The difference in "int" sizes in different C implementations can cause portability problems because of Standard C's integral promotion rule. This problem is getting worse with the introduction of 64-bit architected machines in the industry. This proposal creates a new <inttypes.h> header that defines a set of integer types whose definition and behaviour are consistent in all implementations. This will facilitate the development of portable programs in Standard C.

Proposal: Attached.

Extended Integers For C

5

John W Kwan
Hewlett-Packard Company
Cupertino, California
+1 408 447 6442
email : jkwan@cup.hp.com

Acknowledgement

The idea of `<inttypes.h>` grew out of discussions in an industry group originally formed to address the problem of mapping the `int` data type in 64-bit machines. Many people participated in that group and contributed ideas for such a header, including representatives from Amdahl, Apple, DEC, Hal Computer, HP, IBM, SGI/Mips, Sun and SPARC International. Additional ideas came from members of the Numerical C Extension Group (NCEG) of X3J11 and other members of X3J11. In particular, Ron Guilmette, Tom MacDonald, Frank Farance, Rex Jaeschke, David Keaton, Jim Thomas, Fred Tydeman, Jeffery Zeeb and Jonathan Ziebell reviewed earlier versions of this paper and provided valuable suggestions. Special thanks to Randy Meyers of Digital Equipment Corporation who provided the macros for the formatted I/O specification and who collaborated with me on an earlier version of this paper.

1. Introduction

The C Standard specifies that the language should support four, signed and unsigned, integer data types, `char`, `short`, `int` and `long`. However, the Standard places very little requirement on their size (number of bits) other than that `int` and `short` be at least 16-bits and `long` be at least as long as `int` and not smaller than 32-bits. Traditionally (i.e. under Kernighan and Ritchie), C had always assumed that `int` is the most efficient (i.e. the fastest) integer data type on a machine, and Standard C, with its integral promotion rule, tacitly continues this assumption. For 16-bit systems, most implementations assign 8, 16, 16 and 32 bits to `char`, `short`, `int`, and `long`, respectively. For 32-bit systems, the common practice is to assign 8, 16, 32 and 32 bits to these types. This difference in `int` size can create some interesting problems for users who migrate from one system to another which assigns different sizes to integral types, because Standard C's integral promotion rule can produce silent changes unexpectedly.

Consider the following example :

```
main()
{
        long L = -1;
        unsigned int i = 1;
        if (L > i)
                printf ("L greater than i\n") ;
        else
                printf ("L not greater than i\n") ;
}
```

Under the Standard's promotion rule, this program will print "L greater than i" if size of `int` equals size of `long`; but it will print "L not greater than i" if size of `int` is less than size of `long`. Both results are conforming and correct. Hence the size of the `int` data type is significant in any C implementation. To complicate matters further, the need for an integer type larger than 32 bits arises for those 32-bit systems that support large files. Implementors that feel such a need have introduced a larger integer type commonly referred to as `long long`.

The need for a general solution for the "extended integer" problem has increased with the introduction of 64-bit based systems in the industry. Efforts to find a common `int` size on 64-bit systems have turned out to be much more difficult than expected.

First of all, any change in the size of `int` from the current definition will produce incompatibility; and no mapping of the base integer types to a particular range of values produces satisfactory performance in all systems. A data model that is optimal for one architecture is usually sub-optimal for another. After much discussion, the industry remains divided. However, the current system of different `int` sizes on different platforms makes life difficult for software developers who, as a result, must maintain different source for different machines (usually by using `#ifdef`). This is not very desirable. Providing the means for users to write portable code is a must if Standard C is to become "the programming language of choice."

This specification addresses this very issue. To help software developers write portable code, implementations should provide a set of integer types whose definitions are consistent across machines and independent of operating systems and other implementation idiosyncrasies. This can be provided through a new header called `<inttypes.h>`. This header defines, via `typedef`, integer types of various sizes; implementations are free to `typedef` them to integer types, including well defined extensions, that they support.

7

2. <inttypes.h>

```
/********************** Basic integer types *****************************
**
** The following defines the basic fixed-size integer types.
**
** Implementations are free to typedef them to Standard C integer types or
** extensions that they support. If an implementation does not support one
** of the particular integer data types below, then it should not define the
** typedefs and macros corresponding to that data type.
**
** intmax_t and uintmax_t are to be the longest (in number of bits) signed
** and unsigned integer types supported by the implementation.
**
** intptr_t and uintptr_t are signed and unsigned integer types large enough
** to hold any data pointer; that is, data pointers can be assigned into or
** from these integer types without losing precision.
**
** intfast_t and uintfast_t are the most efficient signed and unsigned
** integer types of the implementation. These shall be at least 16-bits
** long.
**
*/

typedef ? int8_t;        /* 8-bit signed integer */
typedef ? int16_t;       /* 16-bit signed integer */
typedef ? int32_t;       /* 32-bit signed integer */
typedef ? int64_t;       /* 64-bit signed integer */

typedef ? uint8_t;       /* 8-bit unsigned integer */
typedef ? uint16_t;      /* 16-bit unsigned integer */
typedef ? uint32_t;      /* 32-bit unsigned integer */
typedef ? uint64_t;      /* 64-bit unsigned integer */

typedef ? intmax_t;      /* largest signed integer supported */
typedef ? uintmax_t;     /* largest unsigned integer supported */

typedef ? intptr_t;      /* signed integer type capable of holding a void * */
typedef ? uintptr_t      /* unsigned integer type capable of holding a void * */

typedef ? intfast_t;     /* most efficient signed integer type */
typedef ? uintfast_t     /* most efficient unsigned integer type */
```

8

```
/************************ Extended integer types ************************
**
** The following define the smallest integer types that can hold the
** specified number of bits.
**
** Implementations are free to typedef them to Standard C integer types or
** any supported extensions.
*/

/* smallest signed integer of at least 8 bits */
typedef ? int_least8_t;

/* the most efficient signed integer of at least 8 bits */
typedef ? int_fast8_t;

/* smallest signed integer of at least 16 bits */
typedef ? int_least16_t;

/* the most efficient signed integer of at least 16 bits */
typedef ? int_fast16_t;

/* smallest signed integer of at least 32 bits */
typedef ? int_least32_t;

/* the most efficient signed integer of at least 32 bits */
typedef ? int_fast32_t;

/* smallest signed integer of at least 64 bits */
typedef ? int_least64_t;

/* the most efficient signed integer of at least 64 bits */
typedef ? int_fast64_t;

/* smallest unsigned integer of at least 8 bits */
typedef ? uint_least8_t;

/* the most efficient unsigned integer of at least 8 bits */
typedef ? uint_fast8_t;

/* smallest unsigned integer of at least 16 bits */
typedef ? uint_least16_t;

/* the most efficient unsigned integer of at least 16 bits */
typedef ? uint_fast16_t;

/* smallest unsigned integer of at least 32 bits */
typedef ? uint_least32_t;

/* the most efficient unsigned integer of at least 32 bits */
typedef ? uint_fast32_t;

/* smallest unsigned integer of at least 64 bits */
typedef ? uint_least64_t;
```

```
/* the most efficient unsigned integer of at least 64 bits */
typedef ? uint_fast64_t;


/* *************************** limits ***********************************
**
** The following defines the limits for the above types.
**
** INTMAX_MIN (minimum value of the largest supported signed integer type),
** INTMAX_MAX (maximum value of the largest supported signed integer type),
** and UINTMAX_MAX (maximum value of the largest supported unsigned integer
** type) can be set to implementation defined limits.
**
** NOTE : A programmer can test to see whether an implementation supports
** a particular size of integer by testing if the macro that gives the
** maximum for that datatype is defined. For example, if #ifdef UINT64_MAX
** tests false, the implementation does not support unsigned 64 bit integers.
**
** The values used below are merely examples. Actual limits for
** an implementation may vary.
**
** The type of these macros is intentionally unspecified.
**
*/

#define INT8_MIN  (-128)
#define INT16_MIN (-32767-1)
#define INT32_MIN (-2147483647-1)
#define INT64_MIN (-9223372036854775807-1)

#define INT_LEAST8_MIN ?   /* implementation defined */
#define INT_LEAST16_MIN ?  /* implementation defined */
#define INT_LEAST32_MIN ?  /* implementation defined */
#define INT_LEAST64_MIN ?  /* implementation defined */

#define INT_FAST8_MIN ?    /* implmentation defined */
#define INT_FAST16_MIN ?   /* implmentation defined */
#define INT_FAST32_MIN ?   /* implmentation defined */
#define INT_FAST64_MIN ?   /* implmentation defined */

#define INT8_MAX  (127)
#define INT16_MAX (32767)
#define INT32_MAX (2147483647)
#define INT64_MAX (9223372036854775807)

#define UINT8_MAX  (255U)
#define UINT16_MAX (65535U)
#define UINT32_MAX (4294967295U)
#define UINT64_MAX (18446744073709551615U)

#define INTMAX_MIN ?    /* implementation defined */
#define INTMAX_MAX ?    /* implementation defined */
#define UINTMAX_MAX ?   /* implementation defined */
```

10

```
#define INTFAST_MIN ?    /* implementation defined */
#define INTFAST_MAX ?    /* implementation defined */
#define UINTFAST_MAX ?   /* implementation defined */


#define INT_LEAST8_MAX ?     /* implementation defined */
#define INT_LEAST16_MAX ?    /* implementation defined */
#define INT_LEAST32_MAX ?    /* implementation defined */
#define INT_LEAST64_MAX ?    /* implementation defined */

#define INT_FAST8_MAX ?      /* implementation defined */
#define INT_FAST16_MAX ?     /* implementation defined */
#define INT_FAST32_MAX ?     /* implementation defined */
#define INT_FAST64_MAX ?     /* implementation defined */


/* The following 2 macros are provided for testing whether the types
** intptr_t and uintptr_t (integers large enough to hold a void *) are
** defined in this header. They are needed in case the architecture can't
** represent a pointer in any standard integral type.
*/
#define INTPTR_MAX
#define UINTPTR_MAX




/* ************************ CONSTANTS *********************************
**
** The following macros create constants of the above types. The intent is
** that:
**    Constants defined using these macros have a specific size and
**    signedness. The suffix used for int64_t and uint64_t (ll and ull)
**    are for examples only. Implementations are permitted to use other
**    suffixes.
*/

#define __CONCAT__(A,B) A ## B

#define INT8_C(c)          (c)
#define INT16_C(c)         (c)
#define INT32_C(c)         (c)
#define INT64_C(c)         __CONCAT__(c,ll)

#define UINT8_C(c)         __CONCAT__(c,u)
#define UINT16_C(c)        __CONCAT__(c,u)
#define UINT32_C(c)        __CONCAT__(c,u)
#define UINT64_C(c)        __CONCAT__(c,ull)

#define INTMAX_C(c)        __CONCAT__(c,ll)
#define UINTMAX_C(c)       __CONCAT__(c,ull)
```

11

```
/*************************** FORMATTED I/O ***********************************
**
** The following macros can be used even when an implementation has not
** extended the printf/scanf family of functions.
**
** The form of the names of the macros is either "PRI" for printf specifiers
** or "SCN" for scanf specifiers, followed by the conversion specifier letter
** followed by the datatype size. For example, PRId32 is the macro for
** the printf d conversion specifier with the flags for 32 bit datatype.
**
** Separate macros are given for printf and scanf because typically different
** size flags must prefix the conversion specifier letter.
**
*** An example using one of these macros:
**
**    uint64_t u;
**    printf("u = %016" PRIx64 "\n", u);
**
** For the purpose of example, the definitions of the printf/scanf macros
** below have the values appropriate for a machine with 16 bit shorts,
** 32 bit ints, and 64 bit longs.
**
*/

/* printf macros for signed integers */
#define PRId8           "d"
#define PRId16          "d"
#define PRId32          "d"
#define PRId64          "ld"

#define PRIdLEAST8      "d"
#define PRIdLEAST16     "d"
#define PRIdLEAST32     "d"
#define PRIdLEAST64     "ld"

#define PRIdFAST8       "d"
#define PRIdFAST16      "d"
#define PRIdFAST32      "d"
#define PRIdFAST64      "ld"

#define PRIi8           "i"
#define PRIi16          "i"
#define PRIi32          "i"
#define PRIi64          "li"

#define PRIiLEAST8      "i"
#define PRIiLEAST16     "i"
#define PRIiLEAST32     "i"
#define PRIiLEAST64     "li"

#define PRIiFAST8       "i"
#define PRIiFAST16      "i"
#define PRIiFAST32      "i"
```

```
#define PRIiFAST64      "li"

#
#define PRIo8           "o"
#define PRIo16          "o"
#define PRIo32          "o"
#define PRIo64          "lo"

#define PRIoLEAST8      "o"
#define PRIoLEAST16     "o"
#define PRIoLEAST32     "o"
#define PRIoLEAST64     "lo"

#define PRIoFAST8       "o"
#define PRIoFAST16      "o"
#define PRIoFAST32      "o"
#define PRIoFAST64      "lo"

#define PRIx8           "x"
#define PRIx16          "x"
#define PRIx32          "x"
#define PRIx64          "lx"

#define PRIxLEAST8      "x"
#define PRIxLEAST16     "x"
#define PRIxLEAST32     "x"
#define PRIxLEAST64     "lx"

#define PRIxFAST8       "x"
#define PRIxFAST16      "x"
#define PRIxFAST32      "x"
#define PRIxFAST64      "lx"

#define PRIX8           "X"
#define PRIX16          "X"
#define PRIX32          "X"
#define PRIX64          "lX"

#define PRIXLEAST8      "X"
#define PRIXLEST16      "X"
#define PRIXLEST32      "X"
#define PRIXLEST64      "lX"

#define PRIXFAST8       "X"
#define PRIXFAST16      "X"
#define PRIXFAST32      "X"
#define PRIXFAST64      "lX"

/* printf macros for unsigned integers */
#define PRIu8           "u"
#define PRIu16          "u"
#define PRIu32          "u"
#define PRIu64          "lu"
```

13

```
#define PRIuLEAST8       "u"
#define PRIuLEAST16      "u"
#define PRIuLEAST32      "u"
#define PRIuLEAST64      "lu"

#define PRIuFAST8        "u"
#define PRIuFAST16       "u"
#define PRIuFAST32       "u"
#define PRIuFAST64       "lu"

/* scanf macros */
#define SCNd16           "hd"
#define SCNd32           "d"
#define SCNd64           "ld"

#define SCNi16           "hi"
#define SCNi32           "i"
#define SCNi64           "li"

#define SCNo16           "ho"
#define SCNo32           "o"
#define SCNo64           "lo"

#define SCNu16           "hu"
#define SCNu32           "u"
#define SCNu64           "lu"

#define SCNx16           "hx"
#define SCNx32           "x"
#define SCNx64           "lx"

/* The following macros define I/O formats for intmax_t and uintmax_t. */

#define PRIdMAX  ?       /* implementation defined */
#define PRIoMAX  ?       /* implementation defined */
#define PRIxMAX  ?       /* implementation defined */
#define PRIuMAX  ?       /* implementation defined */

#define SCNiMAX  ?       /* implementation defined */
#define SCNdMAX  ?       /* implementation defined */
#define SCNoMAX  ?       /* implementation defined */
#define SCNxMAX  ?       /* implementation defined */
```

14

```
/* The following macros define I/O formats for intfast_t and uintfast_t. */

#define PRIdFAST  ?       /* implementation defined */
#define PRIoFAST  ?       /* implementation defined */
#define PRIxFAST  ?       /* implementation defined */
#define PRIuFAST  ?       /* implementation defined */

#define SCNiFAST  ?       /* implementation defined */
#define SCNdFAST  ?       /* implementation defined */
#define SCNoFAST  ?       /* implementation defined */
#define SCNxFAST  ?       /* implementation defined */


/*************** conversion functions *******************************
**
** The following routines convert from strings to the largest supported
** integer types. They parallel the strtol and strtoul functions in Standard
** C. Implementations are free to equate them to any existing functions
** they may have.
*/

extern intmax_t strtoimax (const char *, char**, int);
extern uintmax_t strtoumax (const char *, char**, int);


/* end of inttypes.h */
```

15

3. Notes

The intent of this paper is to take a "minimalist" approach to solving the extended integer problem in C; one that accommodates, but does not necessarily requires, extensions to the language. Internal representation of a data type (e.g. endianess, bit and byte ordering) is outside the scope of this specification.

Besides defining integer data types of 8, 16, 32 and 64 bits, this header also defines integer types of at least 8, 16, 32 and 64 bits. This is mainly for systems whose word size does not fit the 16-bit or 32-bit word model. Implementations do not have to support all data types defined in this specification.

The various MIN/MAX macros define the limits of each data type. By using the `#ifdef` directive, users can test to see if certain data types are supported in an implementation. For example, if `#ifdef` INT64_MAX tests false, this header does not define 64-bit integers.

The __CONCAT__ macro provides a means to construct constants of a particular type.

The types `intfast_t` and `uintfast_t` define the most efficient signed and unsigned integer type of an implementation. They shall be at least 16-bits long. For those systems that have more than one "most efficient" integer types, the largest one should be used.

This paper also presents a method to handle formatted I/O that requires little or no extension to the language by providing macros for existing formats like `d`, `i`, `o` and `x`. There are no macros corresponding to the `c` conversion specifier. Most implementations do not support the `c` conversion specifier for anything besides `int`. Likewise, there are no `scanf` macros for the 8-bit datatypes as most implementations do not support reading a `char` of any size.

16