N381

X3J11/94-066

Extended Integers For C

Version 3.0

John W Kwan
Hewlett-Packard Company
Cupertino, California

1. Introduction

The ANSI/ISO C standard specifies that the language should support 4 integer data types, `char`, `short`, `int` and `long`. However, the Standard places no requirement on their length (number of bits) other than that `int` be at least as long as `short`, and `long` be at least as long as `int`. 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 the ANSI Standard, with its integral promotion rule, tacitly continues this assumption. For 16-bit systems, most implementations assigned 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 the C Standard's 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 legal and correct. Hence the size of the `int` data type is significant in any C implementation. With the introduction of 64-bit systems in the industry, the choice of the size of `int` is even more important, for it has compatibility as well as performance ramifications.

To complicate matters further, the need for an integer type larger than 32 bits arises for those 32-bit systems that support large files. For those systems that feel a need to have a larger integer type, a new 64-bit integer type commonly referred to as `long long` has been implemented.

The `long long` type was created specifically to satisfy the need for an integer type larger than 32 bits. It is non-standard and therefore non-portable. It was never intended to be a general solution for the "extended integer" problem. 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. Any one 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 must maintain different source for different machines (usually by using #ifdef). This is not very desirable. We must provide the means for users to write portable code if C is to become "the programming language of choice."

This proposal 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 will define, via typedefs, integer types of various sizes; implementations are free to typedef them to integer types, including extensions, that they support. By using this header and the types it provides, developers will be able to use a certain integer type and be assured that it will have the same properties and behaviour on different machines. This header can be easily implemented with minimal or no extensions to the language.

2. <inttypes.h>

```
#ifndef __inttypes_included
#define __inttypes_included

/*********************** Basic integer types ****************************
**
** The following defines the basic fixed-size integer types.
**
** Implementations are free to typedef them to 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 datatype.
**
** intmax_t and uintmax_t are guaranteed to be the largest signed and
** unsigned integer types supported by the implementation.
**
** intptr_t and uintptr_t are signed and unsigned integer types large enough
** to hold a pointer; that is, 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.
**
*/

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 */
```

```
/*********************** Extended integer types ************************
**
** The following defines smallest integer types that can hold n bits.
**
** Implementations are free to typedef them to C integer types or any
** supported extensions.
**
*/

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

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

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

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

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

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

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

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

```
/* **************************** limits ************************************
**
** The following defines the limits for the above types.
**
** INTMAX_MIN (minimum value of the largest supported integer type),
** INTMAX_MAX (maximum value of the largest supported 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 seeing 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 using the 2's complement
** numbering system. Actual limits for an implementation may vary.
**
*/

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

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

#define UINT8_MAX (255)
#define UINT16_MAX (65535)
#define UINT32_MAX (4294967295)

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

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

#define INT64_MIN (-9223372036854775807-1)
#define INT64_MAX (9223372036854775807)
#define UINT64_MAX (18446744073709551615)
```

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

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

#define INT8_C(c)         (c)
#define UINT8_C(c)        __CONCAT__(c,u)

#define INT16_C(c)        (c)
#define UINT16_C(c)       __CONCAT__(c,u)

#define INT32_C(c)        (c)
#define UINT32_C(c)       __CONCAT__(c,u)

#define INT64_C(c)        __CONCAT__(c,ll)
#define UINT64_C(c)       __CONCAT__(c,ull)

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

```
/************************* 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 printf versus scanf macros are given because typically different
** size flags must prefix the conversion specifier letter.
**
** There are no macros corresponding to the c conversion specifier. These
** macros only support what can be done without extending printf/scanf, and
** most implementations do not support the c conversion specifier for
** anything besides int.
**
** Likewise, there are no scanf macros for the 8 bit datatypes. Most
** implementations do not support reading 8 bit integers.
**
** 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.
**
*/
#define PRId8           "d"
#define PRId16          "d"
#define PRId32          "d"
#define PRId64          "ld"

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

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

#define PRIu8           "u"
#define PRIu16          "u"
#define PRIu32          "u"
#define PRIu64          "llu"

#define PRIx8           "x"
#define PRIx16          "x"
```

```
#define PRIx32          "x"
#define PRIx64          "llx"

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

#define SCNd16          "hd"
#define SCNd32          "d"
#define SCNd64          "lld"

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

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

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

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

#define SCNX16          "hX"
#define SCNX32          "X"
#define SCNX64          "llX"

/* The following macros define I/O formats for intmax_t and uintmax_t.
** Their particular values are implementation defined.
*/
#define PRIdMAX  ?
#define PRIoMAX  ?
#define PRIxMAX  ?
#define PRIuMAX  ?

#define SCNiMAX  ?
#define SCNdMAX  ?
#define SCNoMAX  ?
#define SCNxMAX  ?
```

```
/* The following macros define I/O formats for intfast_t and uintfast_t.
** Their particular values are implementation defined.
*/
#define PRIdFAST  ?
#define PRIoFAST  ?
#define PRIxFAST  ?
#define PRIuFAST  ?

#define SCNiFAST  ?
#define SCNdFAST  ?
#define SCNoFAST  ?
#define SCNxFAST  ?
```

```
/*************** conversion functions ********************************
**
** The following routines are proposed to do conversions from strings to the
** largest supported integer types. They parallel the ANSI strto* functions.
** 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);

#endif /* __inttypes_included */

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

3. Notes

The intend of this paper is to take a "minimalist" approach to solving the extended integer problem in C; one that requires little or no extensions to the language. Internal representation of a data type (e.g. endianess, bit and byte ordering) is outside the scope of this paper and proposal.

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 typedef'ed here.

The various MIN/MAX macros define the limits of each data type. They also can be used as a test to see if certain data types are not supported in an implementation by using the #ifdef directive. For example, if #ifdef INT64_MAX tests false, the implementation does not support 64 bit integers. In general, it is expected that most of the integer types defined in this header will be supported; 64 bit types are probably the only exception.

The __CONCAT__ macro provides a means to construct constants of a particular type. The suffix used to denote 64 bit integers, ll and ull, are not standard and are used here only as examples.

For those systems that have more than one "most efficient" integer types, `intfast_t` and `uintfast_t` should be typedef'd to the larger integer.

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`. The only new formats added are for 64 bit integers, `lld`, `llo`, `llx` etc. These are not standard but are common extensions available in many implementations.

4. Acknowledgement

The idea of <inttypes.h> grew out of discussions in an industry committee originally formed to address the problem of `int` data type in 64-bit machines. Many people participated in that committee 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. Special thanks to Randy Meyers of Digital Equipment Corporation who suggested and provided the macros for the formatted I/O proposal and who collaborated with me on an earlier version of this paper.