Document Number: WG14 N420/X3J11 95-021

C9X Revision Proposal

Title: ''big integer'' (at least 64-bit) routines and types
Author: JR (John Rogers)
Author Affiliation: freelance writer
Postal Address: 11604 104th Ave. NE; Kirkland WA 98034; USA
E-mail Address: 72634.2402@CompuServe.com
Telephone Number: +1 206 8212816
Fax Number: _____
Sponsor: _____
Date: 1995-02-07
Proposal Category:
- N_ Editorial change/non-normative contribution
- N_ Correction
- Y_ New feature
- N_ Addition to obsolescent feature list
- N_ Addition to Future Directions
- N_ Other (please specify) _____

Area of Standard Affected:
- N_ Environment
- N_ Language
- N_ Preprocessor
- Y_ Library
  - Y_ Macro/typedef/tag name
  - Y_ Function
  - Y_ Header
- N_ Other (please specify) _____

Prior Art: As of this writing (1995-02-07), I am still developing my implementation of the bigint library. I am not aware of any other implementations of it yet, although I am encouraging others. My implementation is intended to be maximally portable to current ANSI/ISO C compilers. I can envision the compiler vendors adding their own versions of the bigint library for maximal speed on their platforms, but this would not be required.

Target Audience: systems programmers (for compilers, debuggers, assemblers, linkers); numerical programmers (for encryption/decryption); financial programmers.

Related Documents (if any): ''Draft Standard for Big Integer Routines for the C Programming Language'', by me, draft 0.1, attached. All clause numbers in this cover sheet refer to clauses in the big integer draft standard.

This proposal is also consistent with, but does not depend on, IEEE Std 695-1990, ''IEEE Standard for Microprocessor Universal Format for Object Modules'' (MUFOM). The benefits of this proposal are not limited to MUFOM users.

Proposal Attached: Y_ Yes __ No, but what's your interest?

Abstract: I propose optional support for ''big integers'' (those with at least 64 bits of precision). This proposal

only involves new libraries and headers; the compiler need not be changed. For a given implementation, a ''big integer'' would be a fixed size. This avoids the issue of what to do when running out of memory for ''infinite precision integers'', which plagues the Multiple Precision (MP) library provided with some versions of UNIX.

Proposal:

Problem Statement: The current ANSI/ISO C standards only guarantee a minimum of 32 bits of precision in the largest integral types. In many financial applications that is not enough range, and floating point is not precise enough to keep the auditors happy.

Also, we are moving from an era of 32-bit computers to an era of 64-bit computers. While 32-bit machines still dominate, this is a good opportunity to promote cross-development tools (from 32-bit to 64-bit systems). At times these tools need to use different byte ordering, byte size, and integer representation than the native system uses. Personally, I hope to write a simulator of Donald Knuth's new 64-bit computer (MMIX) which would run on any 32-bit computer.

Conceptual Model: Big integers are conceptually like the largest integer types in a given implementation of C, with twice the precision of the ''long integer'' type. However, big integers would be implemented by a library of routines rather then the compiler knowing how to generate code for them. There are routines to do math, shifts, AND/OR/XOR, copy, and get/set bits of big integers. Big integers may be treated as being in any of the four ''usual representations'' (defined in the big integer draft standard, clause 3 (''Definitions''). For instance, one ''usual representation'' is two's complement. The application may also specify some nonnative ''byte'' order or size.

Semantics: The semantics of big integers and related types appear in clause 3 (''Definitions''), clause 4 (''General Requirements''), clause 5 (''<boolean.h>''), clause 6 (''<sign.h>''), clause 7 (''<rep.h>''), and clause 8 (''<bigint.h>'')

Constraints: A number of general constraints are given in the big integer draft standard in clause 4, ''General Requirements''. Additional (per-routine) constraints are given with the descriptions of the routines in clause 6 (''<sign.h>'') and clause 8 (''<bigint.h>''). Constraints against using most standard C operators with big integers appear at the beginning of clause 8 (''<bigint.h>'').

Behavior: Undefined behavior is allowed in subclause 4.2 (''Bit Numbering''). Indeterminate behavior is not allowed. Implementation-defined behavior is allowed in subclauses 3.2.5 (definition of ''endian''), 3.2.11 (definition of ''minimum addressable unit''), 4.1 (''Allocation and Alignment of Big Integers''), 4.8 (''Signed and Unsigned Integer Representation''), and 7

(``<rep.h>'').

Syntax, Synopsis, Keywords, Tokens, Operators, Directives: The prototype for each routine is given in the big integer draft standard. These appear in clause 6 (``<sign.h>'') and clause 8 (``<bigint.h>'').

Rationale: Big integer support is provided for applications which may need more range than that provided by the largest integral type directly supported by the compiler. For those applications which need to emulate other byte sizes or orders (or integer representations), direct control over these aspects is possible.

Implementation Issues: I have tried to specify this in such a way that it be added on to any current ANSI/ISO C implementation, simply by adding new libraries and headers. The ``routines'' may be functions, macros, or anything else that gives the appearance of parameterized macros. For efficiency, the default ``native'' format is indicated by a NULL pointer instead of a pointer to a format. An implementation could detect use of the NULL pointer, and generate inline code via the macro expansion. (For instance, if a C implementation supports ``long long'' or ``quad'' or whatever, that version of ``<bigint.h>'' could take advantage of it.) For non-NULL format pointers, the implementation could call actual functions to do the complicated work using the format.

There are no internationalization issues with the big integer library as currently specified. Someone has suggested revising the big integer specification to comply with ISO/IEC 10967-1, which is part one of the Language Independent Arithmetic (LIA) standard. The LIA-1 standard requires that ``termination with message'' be allowed as one means of notification of ``undefined'' behavior. This change would introduce an internationalization issue, for the message would need to be supplied by the implementation in the correct language.

Language Compatibility Issues: None I am aware of.

Subsetting: I think it is important that programs be able, at compile time, to tell if the big integer library is available. Some predefined preprocessor symbol like ``_HAS_BIG_INTEGERS'' would be good. I'm certainly open to other techniques or symbol names for this.

JR (John Rogers)
11604 104th Ave. NE
Kirkland WA 98034-6606
Internet: 72634.2402@CompuServe.com

Date: 1995-01-04

Subject: Draft standard (0.1) for big integer routines for C

To:

Geoff Baldwin (IEEE MUFOM committee)
Stephen L. Diamond (IEEE CS MM Standards Committee)
Jonathan Erickson (DDJ Editor-in-Chief)
Torbjorn Granlund (GNU Multiple Precision author)
Rex Jaeschke (NCEG)
Burt Kaliski (RSA Laboratories)
Donald Knuth (MMIX designer)
Kevin Leary (Analog Devices Inc.)
John Gerard Malecki (VLSI Libraries Inc.)
Bill Parrette (computer book author)
Tom Pittman (IEEE MUFOM committee)

Hi all!

I have written the first draft of a proposed standard for big integer math routines for C. A big integer would provide at least 64 bits of precision. I used to call this the math64 spec, but "big integer" is more general.

Would you be so kind as to take a look at this draft?

To become involved with the big integer standard, just send me a letter or some email. Please feel free to pass this draft around, publish it, post it to Usenet, etc.

Thanks in advance!

JR

JR (John Rogers)

Draft Standard for Big Integer Routines for the C Programming Language

This is a draft standard, subject to change. For more information contact the author:

JR (John Rogers)
11604 104th Ave. NE
Kirkland WA 98034-6606
Internet: 72634.2402@CompuServe.com

Introduction

(This introduction is not a part of this standard.)

This draft standard defines a set of big integer routines for use with the C programming language. This draft may evolve into an IEEE standard.

The big integer routines defined here may be useful in:

- assemblers
- loaders
- linkers
- compilers
- program librarians (for object modules)
- debuggers
- instruction set simulators
- encryption/decryption

This draft standard relates to the "IEEE Standard for Microprocessor Universal Format for Object Modules" (MUFOM). The MUFOM file format gives a machine-independent way of representing object code, external references, etc. MUFOM allows 64 bits for addresses and some other expressions. Unfortunately, ANSI C only guarantees a minimum of 32 bits of precision for the largest integer types. A logical stepping stone to the MUFOM routines is a way of representing and computing with (at least) 64-bit precision integers in C. These big integer routines may become a required part of some MUFOM routines someday. That is why these big integer routines require at least 64 bits of precision.

This standard provides a common solution to these and other (at least) 64-bit integer computation requirements. It defines a set of "big integer" routines for C. These routines allow the caller to specify whether to use two's complement, one's complement, unsigned, or signed-magnitude representation. They also allow the caller to specify a desired bit order (little-endian or big-endian). For callers who aren't interested in a particular representation or bit order, a way to request "native" (presumably the fastest) facilities is provided.

Contents

1. Overview

This standard specifies a set of routines for using big (at least 64-bit wide) integers in C programs.

This standard does not specify:

a) the algorithms to be used by a conforming implementation
b) the actual order of bits within a big integer when stored in memory

2. References

This standard shall be used in conjunction with the following publications. When the following standards are superseded by an approved revision, the revision shall apply.

ANSI X3.159-1989, American National Standard for Information Systems -- Programming Language -- C.[1]

IEEE Std 695-1990, IEEE Standard for Microprocessor Universal Format for Object Modules.[2]

3. Definitions

3.1 Terminology

3.1.1 conforming implementation: An implementation that provides at least the headers, data types, and routines specified in this standard. A conforming implementation may have implementation-defined extensions, provided they do not alter the behavior of a conforming program.[3]

3.1.2 conforming program: A program that only uses the mechanisms in the following list to access the data types defined in this standard, and does not require any specific implementation-defined behavior.

---

[1]ANSI publications are available from the American National Standards Institute, 11 West 42nd St., New York, NY 10036, USA.

[2]IEEE publications are available from the Institute of Electrical and Electronic Engineers, 445 Hoes Lane, P.O Box 1331, Piscataway NJ 08855-1331, USA.

[3]Paraphrased from various parts of the C Standard.

) the routines defined in this standard
) for big integer access, the standard C operators given in **4.10 Standard C Operators and Big Integers**
) for data types other than big integer, any applicable standard C operator
) for big integers, any memory allocation and alignment mechanism which meets the requirements in **4.1 Allocation and Alignment of Big Integers**

plementation: A particular set of software, running in a particular translation ıent, that provides headers for, and supports the execution of routines in, a r execution environment.[4]

plementation-defined behavior: Behavior that, for a correct program construct ect data, depends on the characteristics of the implementation and that each ıtation shall document.[5]

y: Is permitted to ..., are permitted to ....

ll: Is required to ..., are required to ....

defined behavior: Behavior for which this standard imposes no requirements.

eral Terms

-endian: A minimum addressable unit (MAU) order where the most significant a larger entity occupies the lowest address, followed by successively less t MAUs of that entity.

integer: A contiguous set of at least 64 bits in memory that meets the ents in the following list.

- has a size of exactly twice the number of bits as the **unsigned long** data type for that implementation of C
- has alignment that meets the requirements in **4.1 Allocation and Alignment of Big Integers**
- may be viewed as having a high part (most significant half) and a low part (least significant half)
- may be interchangebly used in any of the usual representations

---

[4] Phrased the C Standard definition of *implementation* here.
[5] Phrased from the C Standard.

3.2.3 bit: The C Standard defines *bit* as "the unit of data storage in the execution environment large enough to hold an object that may have one of two values." The same definition applies in this standard.

3.2.4 data bits: The numeric value bits (except for the sign bit, if any) of a big integer. The assigning of a numeric value to a particular data bit depends on the layout and the value of the sign bit (if any).

3.2.5 endian: The order of minimum addressable units (MAUs) of a larger entity. A conforming implementation shall support at least big-endian and little-endian MAU orders. Other (implementation-defined) MAU orders may be provided; these are sometimes referred to as middle-endian orders.

3.2.6 high part of a big integer: The most significant half of a big integer. Note that in many representations, the high part of a big integer includes the sign bit as the high bit.

3.2.7 layout: The combination of the kind of integer representation, minimum addressable unit (MAU) size in bits, and MAU ordering ("endian").

3.2.8 little-endian: A minimum addressable unit (MAU) order where the least significant MAU of a larger entity occupies the lowest address, followed by successively more significant MAUs of that entity.

3.2.9 low part of a big integer: The data bits of a big integer that are numbered 0 through ((size of the big integer in bits)/2)-1. (See **4.2 Bit Numbering** for the rules on numbering bits.) This is the least significant half of a big integer.

3.2.10 middle-endian: Any of a number of minimum addressable unit (MAU) orders that are neither big-endian nor little-endian.

3.2.11 minimum addressable unit: The MUFOM File Standard defines this as follows: "For a given processor, the amount of memory located between an address and the next address. It is not equivalent to a word or a byte." The same definition applies to this standard. Both standards use the abbreviation MAU for this term.

A conforming implementation shall support at least the minimum addressable unit (MAU) sizes (in bits) given in the following list. A conforming implementation may also support other implementation-defined MAU sizes.

a) 1
b) 2
c) **CHAR_BIT**
d) the size of an **unsigned long** variable (in bits)
e) the size of big integer (in bits)
f) 2 * **CHAR_BIT**

3.2.12 native-endian: The minimum addressable unit (MAU) order that an implementation has defined as its default. Note that the native-endian order may be big-endian, little-endian, or some specific middle-endian order.

3.2.13 native layout: The layout consisting of the following.

a) signless integer representation
b) the native-endian minimum addressable unit (MAU) order
c) the MAU size which that implementation has defined as its default

A conforming implementation shall use its native layout implicitly when a caller has passed a null pointer (to a routine in this standard) in place of a pointer to an explicit layout.

3.2.14 "negative zero": In certain representations, an alternative bit pattern for the "normal zero." For instance, in signed-magnitude representation, a "normal zero" has all data bits of zero and a sign bit of zero; a "negative zero" in that format has all data bits of zero but has a sign bit of one. A conforming implementation shall support "negative zero" in at least the one's complement and signed-magnitude representations.

3.2.15 pure binary numeration system: A "positional representation for integers that uses the binary digits 0 and 1, in which the values represented by successive bits are additive, begin with 1, and are multiplied by successive integral powers of 2, except perhaps the bit with the highest position."[6]

3.2.16 representation: Any of the various methods for assigning integral (possibly signed) values to a set of bits in a given number.

3.2.17 routine: A function-like entity defined in this standard. This standard documents each routine by giving the C syntax to declare it as a function. For a given implementation, a routine may be (among other things) a C function or a function-like macro. The technique by which a header provides access to a routine is unspecified. Note that, unlike an actual C function, a routine may evaluate its arguments more than once.

3.2.18 sign bit: A bit (within certain integer representations) which indicates the sign of an integer. The exact meaning of the sign bit, and how it affects the data bits for that representation, depends on the representation.

3.2.19 signless: A representation in which all of the bits are data bits. This is often referred to as "unsigned" outside this standard, but this could be confused with the C keyword **unsigned**. So, this standard uses the term "signless" to refer to the representation.

---

[6]This is quoted from the C Standard.

3.2.20 usual representations: The four common representations for integers: two's complement, one's complement, signed-magnitude and signless. A conforming implementation shall support at least the usual representations.

3.3 Abbreviations

3.3.1 C Standard: ANSI X3.159-1989, American National Standard for Information Systems -- Programming Language -- C, or latest revision thereof.

3.3.2 MAU: Minimum addressable unit.

3.3.3 MAUs: Minimum addressable units.

3.3.4 MUFOM: Microprocessor Universal Format for Object Modules.

3.3.5 MUFOM File Standard: IEEE Std 695-1990, IEEE Standard for Microprocessor Universal Format for Object Modules, or latest revision thereof.

4. General Requirements

4.1 Allocation and Alignment of Big Integers

A conforming implementation shall support at least the techniques in the following list for allocating memory for a big integer (the Big_Int_T type).

a) using the standard C library functions **calloc**, **malloc**, and **realloc**, each with a request for **sizeof**(Big_Int_T)
b) defining an automatic storage duration (either "plain" or with the **auto** keyword) Big_Int_T variable in any block
c) defining a **static** Big_Int_T variable (whether inside a block or not)
d) defining an external linkage Big_Int_T variable ("plain" or **static**)

A conforming implementation may also provide other implementation-defined memory allocation techniques.

A conforming implementation shall not require stricter alignment for big integers than that provided by the use of any of the techniques in the list above.

4.2 Bit Numbering

Some routines defined in this standard receive a bit number as an argument. Consistent with the MUFOM File Standard, this standard numbers bits such that the least significant bit has the number zero.

Except for shift/rotate counts, the effect of invoking a routine defined in this standard with a bit number larger than the size of a big integer is undefined. See **4.7 Shift/Rotate Counts** and **4.9 Size of a Big Integer**.

4.3 Headers

Each conforming implementation shall provide **<bigint.h>** and the other C headers defined in this standard. A conforming program may include these headers in any order. A conforming program may include each of these headers more than one once in any given translation unit.

4.4 Ordering of Big Integer Bits

The big integer routines defined by this standard shall process big integers as if they have the layouts indicated. However, a conforming implementation may or may not actually store the bits in memory in the order indicated.

4.5 Parameter Restrictions

A conforming program shall not provide pointers to the same big integer as two or more arguments to any call to a routine documented in this standard.

Unlike an actual C function, any routine defined by this standard may evaluate its arguments more than once. A conforming program shall provide any argument (to any routine defined in this standard) which causes a side-effect when the argument is evaluated.

4.6 Reserved Names

A conforming program shall treat certain groups of identifiers as being reserved. The table below lists them. The preprocessor identifiers are only reserved if a header defined in this standard is included.

name space	name(s)	who this is reserved for
external	prefix Big[A-Z]	reserved for future standards
external	prefix Big_	reserved for implementations
preprocessor	prefix BIG_	reserved for future standards

preprocessor	BIGINT_H	reserved for implementations
preprocessor	BOOLEAN_H	reserved for implementations
preprocessor	REP_H	reserved for implementations
typedef	suffix _T	reserved for future standards
preprocessor	SGN_H	reserved for implementations

Note: In the above table, the notation "[A-Z]" indicates an upper-case letter.

4.7 Shift/Rotate Counts

The various routines that use shift/rotate counts are BigRoL, BigRoR, BigShL, and BigShR. These routines shall implement the semantics of those counts as follows:

shift/rotate count	semantics
0	no shift
1 to (size in bits of big integer-1)	shift/rotate normally
>= (size in bits of big integer)	no shift

See also **4.9 Size of a Big Integer**.

4.8 Signed and Unsigned Big Integer Representation

A conforming implementation shall support at least the four following usual representations for big integers. Numbers in the usual representations shall obey the "pure binary numeration system" definition given in this standard; any implementation-defined representations may or may not obey that definition.

An Int_Rep_T value of NoSign shall indicate a number in signless representation. In this representation, there shall be no sign bit; all of the bits of the big integer shall be data bits.

An Int_Rep_T value of OnesComplement shall indicate a number in one's complement representation. [Expand]

An Int_Rep_T value of SignedMagnitude shall indicate a number in signed-magnitude representation. [Expand]

An Int_Rep_T value of TwosComplement shall indicate a number in two's complement representation. [Expand]

4.9 Size of a Big Integer

A conforming program may compute the size (in bits) of a big integer by multiplying **sizeof**(Big_Int_T) by **CHAR_BIT**. In a conforming implementation, this product shall be at least 64. **<limits.h>** defines **CHAR_BIT**, as required by the C Standard. A conforming implementation shall guarantee that there are no padding or "extra" bits in the usual representations.

A conforming program may also compute the size (in characters) of a big integer by using **sizeof**(Big_Int_T).

4.10 Standard C Operators and Big Integers

A conforming program may only use the following standard C operators with big integers (specifically, the Big_Int_T type).

=	(assignment)
==	(comparison for equality)
&	(address of)
sizeof	

5. <boolean.h>

The **<boolean.h>** header shall define an enumerated type (named Boolean_T), which shall have two named values. The values shall be named Boolean_False (equivalent to 0) and Boolean_True (equivalent to 1).

6. <sign.h>

6.1 Type

The **<sign.h>** header shall declare a Sign_T type, which shall be an integral type. **<sign.h>** shall also declare these values of Sign_T, with the values given:

SGN_NEGATIVE	(with a value of -1)
SGN_ZERO	(with a value of 0)
SGN_POSITIVE	(with a value of +1)

Note that the SGN_ZERO value also refers to the "negative zero" condition.

No other values for the Sign_T type shall be used by a conforming program or returned by a conforming implementation.

6.2 Routines

6.2.1 SignMultiply

```
#include <sign.h>
Sign_T
SignMultiply(
        Sign_T  First,
        Sign_T  Second);
```

If either First or Second is SGN_ZERO, then SignMultiply shall return SGN_ZERO. If both First and Second have the same nonzero value, then SignMultiply shall return SGN_POSITIVE. Otherwise, SignMultiply shall return SGN_NEGATIVE.

6.2.2 SignOpposite

```
#include <sign.h>
Sign_T
SignOpposite(
        Sign_T  Original);
```

The SignOpposite routine shall return the Sign_T value given in the following table:

Original	returned value
SGN_NEGATIVE	SGN_POSITIVE
SGN_ZERO	SGN_ZERO
SGN_POSITIVE	SGN_NEGATIVE

7. <rep.h>

The **<rep.h>** header collects the things necessary to describe the layout (including integer representation) of various data types. This standard defines those parts of **<rep.h>** which a conforming program uses to describe integers (including big integers). Other standards may expand this header, for instance to describe the layout of an address to a portable linker.

The **<rep.h>** header shall declare the Int_Rep_T enumerated type, with at least the values in the list below.

NoSign
OnesComplement
SignedMagnitude
TwosComplement

The **<rep.h>** header shall also declare the MAU_Order_T enumerated type, with at least the values in the list below. Other implementation-defined values may be provided. The effect of passing an implementation-defined MAU_Order_T value to one of the routines specified in this standard is also implementation-defined.

BigEndian
LittleEndian

<rep.h> shall also declare the Bit_Number_T type, as some unsigned integral type. The maximum value of Bit_Number_T shall be at least as large as the number of bits in a big integer. Some routines use this type to indicate a bit number, a length in bits, or a shift/rotate count.

The layout of a given big integer is described by the Int_Rep_And_Format_T (integer representation and format) type. For those routines which may be passed a pointer to a layout, a null pointer implicitly refers to the native layout. **<rep.h>** shall declare the Int_Rep_And_Format_T type as a structure containing the following members, in any order:

Type	Member Name
MAU_Order_T	MauOrder
Int_Rep_T	IntRep
Bit_Number_T	BitsPerMau

8. <bigint.h>

The **<bigint.h>** header shall include the **<boolean.h>**, **<rep.h>**, **<sign.h>**, and **<stdio.h>** headers.

8.1 Type

The Big_Int_T type is the actual big integer. Once memory has been allocated for the big integer, the big integer does not need to be specially initialized. (No particular initial value is guaranteed, however.)

The **<bigint.h>** shall declare the Big_Int_T type. Each implementation may declare this type differently, and a conforming implementation need not document how it has implemented the type.

Except for the standard C operators listed in **4.10 Standard C Operators and Big Integers**, a conforming program shall not directly make use of the contents of the Big_Int_T type in any way. In particular, a conforming program shall not use any of the techniques in the following list.

a) access the member(s) (if any) of the structure (if any)
b) cast the Big_Int_T data type to another data type
c) cast a pointer to a Big_Int_T data type to another data type
d) place it in a union and then access it as a different data type.

A conforming program shall also follow the rules in **4.1 Allocation and Alignment of Big Integers**.

8.2 Routines

8.2.1 BigAbs

```
#include <bigint.h>
Boolean_T BigAbs(
        Big_Int_T * Dest,
        const Big_Int_T * Src,
        const Int_Rep_And_Format_T * Layout);
```

If the absolute value of the big integer at *Src can be represented, then the BigAbs routine shall set the big integer at *Dest to that absolute value, and return Boolean_False to indicate no overflow. Otherwise, BigAbs shall set the big integer at *Dest to the largest possible big integer, and return Boolean_True to indicate an overflow occurred.

8.2.2 BigAdd

```
#include <bigint.h>
Boolean_T BigAdd(
        Big_Int_T * Result,
        const Big_Int_T * A_Number,
        const Big_Int_T * Another_Number,
        Boolean_T Previous_Carry,
        const Int_Rep_And_Format_T * Layout);
```

The BigAdd (big integer add) routine shall set the big integer at *Result to the part of the sum of the big integers *A_Number and *Another_Number (plus one, if Previous_Carry is Boolean_True) that fits in a big integer. If a carry occurred, then BigAdd shall return Boolean_True; otherwise, BigAdd shall return Boolean_False.

8.2.3 BigAnd

```
#include <bigint.h>
void BigAnd(
        Big_Int_T * Result,
        const Big_Int_T * A_Number,
        const Big_Int_T * Another_Number);
```

The BigAnd (big integer AND) routine shall set the big integer at *Result to the bitwise AND of the big integers *A_Number and *Another_Number. The sign bits (if any) are ANDed together, just as the corresponding data bits are, in a bitwise fashion.

8.2.4 BigBClr

```
#include <bigint.h>
void BigBClr(
        Big_Int_T * Dest,
        const Big_Int_T * Src,
        Bit_Number_T BitNum,
        const Int_Rep_And_Format_T * Layout);
```

The BigBClr (big integer bit clear) routine shall set the big integer *Dest to the result of clearing the bit numbered BitNum in the big integer *Src.

8.2.5 BigBFlip

```
#include <bigint.h>
void BigBFlip(
        Big_Int_T * Dest,
        const Big_Int_T * Src,
        Bit_Number_T BitNum,
        const Int_Rep_And_Format_T * Layout);
```

The BigBFlip (big integer bit flip) routine shall set the big integer *Dest to the result of flipping (toggling) the bit numbered BitNum in the big integer *Src.

8.2.6 BigBSet

```
#include <bigint.h>
void BigBSet(
        Big_Int_T * Dest,
        const Big_Int_T * Src,
        Bit_Number_T BitNum,
        const Int_Rep_And_Format_T * Layout);
```

The BigBSet (big integer bit set) routine shall set the big integer *Dest to the result of setting (turning on) the bit numbered BitNum in the big integer *Src.

8.2.7 BigBTest

```
#include <bigint.h>
Boolean_T BigBTest(
        const Big_Int_T * A_Number,
        Bit_Number_T BitNum,
        const Int_Rep_And_Format_T * Layout);
```

The BigBTest (big integer bit test) routine shall examine a bit (indicated by BitNum) in the big integer *A_Number. If that bit is on, BigBTest shall return Boolean_True, otherwise BigBTest shall return Boolean_False.

8.2.8 BigCmp

```
#include <bigint.h>
int BigCmp(
        const Big_Int_T * A_Number,
        const Big_Int_T * Another_Number,
        const Int_Rep_And_Format_T * Layout);
```

The BigCmp routine shall compare the big integer *A_Number to the big integer *Another_Number. (The BigCmp routine shall treat a "negative zero" as being equal to a regular zero.) If A_Number is larger than Another_Number, then BigCmp shall return a value greater than zero. If A_Number is equal to Another_Number, then BigCmp shall return zero. Otherwise, BigCmp shall return a value less than zero.

8.2.9 BigCopy

```
#include <bigint.h>
void BigCopy(
```

```
    Big_Int_T * Dest,
    const Big_Int_T * Src);
```

The BigCopy routine shall set the big integer at *Dest to a copy of the big integer at *Src.

8.2.10 BigDecr

```
#include <bigint.h>
Boolean_T BigDecr(
    Big_Int_T * Dest,
    const Big_Int_T * Src,
    Boolean_T Previous_Borrow,
    const Int_Rep_And_Format_T * Layout);
```

The BigDecr (big integer decrement) routine shall subtract one (two if and only if Previous_Borrow is Boolean_True) from the big integer *Src. BigDecr shall set the big integer *Dest to the part of the result that fits in a big integer. BigDecr shall return Boolean_True if a borrow occurred in the subtraction; otherwise, BigDecr shall return Boolean_False.

8.2.11 BigDiv

```
#include <bigint.h>
void BigDiv(
    Big_Int_T * Quotient,
    Big_Int_T * Remainder,
    const Big_Int_T * Numerator,
    const Big_Int_T * Denominator,
    const Int_Rep_And_Format_T * Layout);
```

If the value of the big integer *Denominator is not zero, then:

> The BigDiv (big integer divide) routine shall set the big integer *Quotient to the integral quotient of dividing *Numerator by *Denominator. BigDiv shall also set the big integer *Remainder to the remainder after dividing *Numerator by *Denominator. BigDiv shall set the sign of *Remainder to the sign of *Denominator. BigDiv shall guarantee that the resulting absolute value of *Remainder is less than the absolute value of *Denominator.

Otherwise, the BigDiv routine shall set *Remainder to zero.

8.2.12 BigExt

```
#include <bigint.h>
void BigExt(
        Big_Int_T * Dest,
        const Big_Int_T * Src,
        Bit_Number_T StartBitNum,
        Bit_Number_T BitCount,
        const Int_Rep_And_Format_T * Layout);
```

The BigExt (big integer extract bits) routine shall extract BitCount bits from the big integer *Src, where StartBitNum is the number of the lowest bit to extract. BigExt shall set the lower BitCount bits of the big integer *Dest to those extracted bits. BigExt shall set the other data bits (if any) and sign bit (if any) of *Dest to zero. If BitCount is zero, then BigExt shall act as if calling BigZero(Dest).

8.2.13 BigHighPart

```
#include <bigint.h>
unsigned long BigHighPart(
        const Big_Int_T * A_Number,
        const Int_Rep_And_Format_T * Layout);
```

The BigHighPart routine shall return the high part of the big integer *A_Number.

8.2.14 BigIncr

```
#include <bigint.h>
Boolean_T BigIncr(
        Big_Int_T * Dest,
        const Big_Int_T * Src,
        Boolean_T Previous_Carry,
        const Int_Rep_And_Format_T * Layout);
```

The BigIncr (big integer increment) routine shall add one (two if and only if Previous_Carry is Boolean_True) to the big integer *Src. BigIncr shall set the big integer *Dest to the part of the result that fits in a big integer. BigIncr shall return Boolean_True if a carry occurred in the addition; otherwise, BigIncr shall return Boolean_False. [What if Layout.IntRep is NoSign and *Src is zero?]

8.2.15 BigIns

```
#include <bigint.h>
```

```
void BigIns(
        Big_Int_T * Dest,
        const Big_Int_T * BitSrc,
        const Big_Int_T * IntSrc,
        Bit_Number_T StartBitSrcBitNum,
        Bit_Number_T StartIntSrcBitNum,
        Bit_Number_T BitCount,
        const Int_Rep_And_Format_T * Layout);
```

The BigIns (big integer insert bits) routine shall extract BitCount bits from the big integer *BitSrc, where StartBitSrcBitNum is the number of the lowest bit to extract. BigIns shall set BitCount bits of the big integer *Dest to those extracted bits, where StartBitSrcBitNum is the number of the lowest bit to set. BigIns shall set the other data bits (if any) and sign bit (if any) of *Dest to the equivalent bits of *IntSrc. If BitCount is zero, then BigIns shall act as if calling BigCopy(Dest, IntSrc).

8.2.16 BigIsOdd

```
#include <bigint.h>
Boolean_T BigIsOdd(
        const Big_Int_T * A_Number,
        const Int_Rep_And_Format_T * Layout);
```

The BigIsOdd routine shall return Boolean_True if the big integer *A_Number is odd; otherwise BigIsOdd shall return Boolean_False.

8.2.17 BigLowPart

```
#include <bigint.h>
unsigned long BigLowPart(
        const Big_Int_T * A_Number,
        const Int_Rep_And_Format_T * Layout);
```

The BigLowPart routine shall return the low part of the big integer *A_Number.

8.2.18 BigMax

```
#include <bigint.h>
void BigMax(
        Big_Int_T * Dest,
        const Big_Int_T * A_Number,
        const Big_Int_T * Another_Number,
```

```
    const Int_Rep_And_Format_T * Layout);
```

The BigMax (big integer maximum) routine shall set the big integer *Dest to the larger of the big integers *A_Number and *Another_Number. If one of the big integers is zero and the other is "negative zero," then it is unspecified which one BigMax shall copy to *Dest.

8.2.19 BigMin

```
#include <bigint.h>
void BigMin(
    Big_Int_T * Dest,
    const Big_Int_T * A_Number,
    const Big_Int_T * Another_Number,
    const Int_Rep_And_Format_T * Layout);
```

The BigMin (big integer minimum) routine shall set the big integer *Dest to the smaller of the big integers *A_Number and *Another_Number. If one of the big integer inputs is zero and the other is "negative zero," then it is unspecified which one BigMin shall copy to *Dest.

8.2.20 BigMod

```
#include <bigint.h>
void BigMod(
    Big_Int_T * Remainder,
    const Big_Int_T * Numerator,
    const Big_Int_T * Denominator,
    const Int_Rep_And_Format_T * Layout);
```

If the value of the big integer *Denominator is not zero, then:

> The BigMod (big integer modulo) routine shall set the big integer *Remainder to the remainder after dividing *Numerator by *Denominator. BigMod shall set the sign of *Remainder to the sign of *Denominator. BigMod shall guarantee that the resulting absolute value of *Remainder is less than the absolute value of *Denominator.

Otherwise, the BigMod routine shall set *Remainder to zero.

8.2.21 BigMul

```
#include <bigint.h>
```

```
Boolean_T BigMul(
        Big_Int_T * Low_Result,
        Big_Int_T * High_Result,
        const Big_Int_T * A_Number,
        const Big_Int_T * Another_Number,
        const Int_Rep_And_Format_T * Layout);
```

The BigMul (big integer multiply) routine shall multiply the big integers *A_Number and *Another_Number. BigMul shall set the big integer *High_Result to the most significant part of the product (including the sign bit, if applicable). BigMul shall set the big integer *Low_Result to the least significant part of the product (including another copy of the sign bit, if applicable). [Make sure all bits of result fit!] If an overflow occurred, BigMul shall return Boolean_True; otherwise BigMul shall return Boolean_False.

8.2.22 BigNeg

```
#include <bigint.h>
Boolean_T BigNeg(
        Big_Int_T * Dest,
        const Big_Int_T * Src,
        const Int_Rep_And_Format_T * Layout);
```

If BigSign(Src,Layout) would return SGN_ZERO, then BigNeg shall act as if calling BigCopy(Dest,Src). (This shall also preserve a "negative zero" value.) BigNeg shall then return Boolean_False to indicate that no underflow/overflow has occurred. [Is this redundant?]

If Layout.IntRep has a value of NoSign, then BigNeg shall act as if calling BigCopy(Dest,Src). BigNeg shall then return Boolean_False to indicate that no underflow/overflow has occurred. [Call BigNot instead?]

If Layout.IntRep has a value of OnesComplement, then BigNeg shall act as if calling BigNot(Dest,Src). BigNeg shall then return Boolean_False to indicate that no underflow/overflow has occurred.

If Layout.IntRep has a value of SignedMagnitude, then BigNeg shall copy the data bits from Src to Dest, and set the sign bit at Dest to the inverted sign bit from Src. BigNeg shall then return Boolean_False to indicate that no underflow/overflow has occurred.

If Layout.IntRep has a value of TwosComplement, then BigNeg shall [expand, including return value...]

8.2.24 BigNot

```
#include <bigint.h>
void BigNot(
        Big_Int_T * Dest,
        const Big_Int_T * Src);
```

The BigNot routine shall set the big integer at *Dest to the result of inverting each of the bits of the big integer at *Src. BigNot shall invert the sign bit (if any) along with the data bits.

8.2.25 BigOr

```
#include <bigint.h>
void BigOr(
        Big_Int_T * Result,
        const Big_Int_T * A_Number,
        const Big_Int_T * Another_Number);
```

The BigOr (big integer bitwise OR) routine shall set the big integer at *Result to the bitwise OR of the big integers *A_Number and *Another_Number. The sign bits (if any) are ORed together, just as the corresponding data bits are, in a bitwise fashion.

8.2.26 BigOut

```
#include <bigint.h>
int BigOut(
        FILE * A_File,
        const Big_Int_T * A_Number,
        const Int_Rep_And_Format_T * Layout,
        unsigned char Base,
        Boolean_T PreferUpperCase);
```

The BigOut (big integer output) routine shall output a printable representation of the big integer *A_Number, to A_File. The results shall be in radix Base (which a conforming program shall only give a value from 2 to 36). [Expand that, including use of Base and the flag.] The BigOut routine shall return the number of characters written to A_File, except in the event of an error, in which case BigOut shall return a negative value.

8.2.27 BigRoL

```
#include <bigint.h>
void BigRoL(
```

```
    Big_Int_T * Dest,
    const Big_Int_T * Src,
    Bit_Number_T ShiftCount,
    const Int_Rep_And_Format_T * Layout);
```

The BigRoL (big integer rotate left) routine shall set the big integer *Dest to the result of rotating the bits (including sign bit, if any) from big integer *Src left by ShiftCount bits. The semantics in **4.7 Shift/Rotate Counts** shall apply.

8.2.28 BigRoR

```
#include <bigint.h>
void BigRoR(
    Big_Int_T * Dest,
    const Big_Int_T * Src,
    Bit_Number_T ShiftCount,
    const Int_Rep_And_Format_T * Layout);
```

The BigRoR (big integer rotate right) routine shall set the big integer *Dest to the result of rotating the bits (including sign bit, if any) from big integer *Src right by ShiftCount bits. The semantics in **4.7 Shift/Rotate Counts** shall apply.

8.2.29 BigShL

```
#include <bigint.h>
void BigShL(
    Big_Int_T * Dest,
    const Big_Int_T * Src,
    Bit_Number_T ShiftCount,
    const Int_Rep_And_Format_T * Layout);
```

The BigShL (big integer shift left) routine shall set the big integer *Dest to the result of shifting the bits (including the sign bit, if any) from big integer *Src left by ShiftCount bits (and shifting-in zeros on the right if necessary). The semantics in **4.7 Shift/Rotate Counts** shall apply.

8.2.30 BigShR

```
#include <bigint.h>
void BigShR(
    Big_Int_T * Dest,
    const Big_Int_T * Src,
```

```
        Bit_Number_T ShiftCount,
        const Int_Rep_And_Format_T * Layout);
```

The BigShR (big integer shift right) routine shall set the big integer *Dest to the result of shifting the bits (including the sign bit, if any) from big integer *Src right by ShiftCount bits (and shifting-in ? on the left if necessary). The semantics in **4.7 Shift/Rotate Counts** shall apply. [What happens to sign bit?]

8.2.31 BigSign

```
#include <bigint.h>
Sign_T BigSign(
        const Big_Int_T * Number,
        const Int_Rep_And_Format_T * Layout);
```

The BigSign routine shall examine the big integer at *Number, test the data bits (and sign bit if applicable), and return the Sign_T value indicated in the following table.

Layout->IntRep	data bits	sign bit	returned value
NoSign	zero	(none)	SGN_ZERO
NoSign	nonzero	(none)	SGN_POSITIVE
OnesComplement	zero	zero	SGN_ZERO
OnesComplement	zero	one	SGN_ZERO
OnesComplement	nonzero	zero	SGN_POSITIVE
OnesComplement	nonzero	one	SGN_NEGATIVE
SignedMagnitude	zero	zero	SGN_ZERO
SignedMagnitude	zero	one	SGN_ZERO
SignedMagnitude	nonzero	zero	SGN_POSITIVE
SignedMagnitude	nonzero	one	SGN_NEGATIVE
TwosComplement	zero	zero	SGN_ZERO
TwosComplement	zero	one	SGN_NEGATIVE
TwosComplement	nonzero	zero	SGN_POSITIVE
TwosComplement	nonzero	one	SGN_NEGATIVE

8.2.32 BigSMul

```
#include <bigint.h>
Boolean_T BigSMul(
        Big_Int_T * Result,
        long A_Number,
        long Another_Number,
        const Int_Rep_And_Format_T * Layout);
```

The BigSMul (signed multiply to a big integer) routine shall multiply the long integer A_Number by Another_Number, and store the result in the big integer *Result. [But what if Layout.IntRep is NoSign?] If an overflow occurred, BigSMul shall return Boolean_True; otherwise BigSMul shall return Boolean_False.

8.2.33 BigSSet

```
#include <bigint.h>
void BigSSet(
        Big_Int_T * Dest,
        long NewValue,
        const Int_Rep_And_Format_T * Layout);
```

The BigSSet (big integer signed set) routine shall set the low order data bits of the big integer *Dest to the bits of NewValue. BigSSet shall also propagate the sign bit to the other data bits (and the sign bit, if applicable) of the big integer *Dest.

[But what if Layout.IntRep is NoSign?]

8.2.34 BigSub

```
#include <bigint.h>
Boolean_T BigSub(
        Big_Int_T * Result,
        const Big_Int_T * A_Number,
        const Big_Int_T * Another_Number,
        Boolean_T Previous_Borrow,
        const Int_Rep_And_Format_T * Layout);
```

BigSub shall set the big integer at *Result to the part of the result of subtracting the big integer *Another_Number (plus one, if Previous_Borrow is Boolean_True) from *A_Number that fits in a big integer. [That wording is ambiguous; fix it!] If a borrow occurred, then BigSub shall return Boolean_True; otherwise, BigSub shall return Boolean_False.

8.2.35 BigSwap

```
#include <bigint.h>
void BigSwap(
        Big_Int_T * Dest,
        const Big_Int_T * Src,
```

```
        const Int_Rep_And_Format_T * NewLayout,
        const Int_Rep_And_Format_T * OldLayout);
```

The BigSwap (big integer swap MAUs) routine shall ?

8.2.36 BigUMul

```
#include <bigint.h>
Boolean_T BigUMul(
        Big_Int_T * Result,
        unsigned long A_Number,
        unsigned long Another_Number,
        const Int_Rep_And_Format_T Layout);
```

The BigUMul (unsigned multiply to a big integer) routine shall multiply the unsigned long integer A_Number by Another_Number, and store the part of the product that fits into the big integer *Result. If an overflow occurred, BigUMul shall return Boolean_True; otherwise BigUMul shall return Boolean_False.

8.2.37 BigUSet

```
#include <bigint.h>
void BigUSet(
        Big_Int_T * Dest,
        unsigned long NewValue,
        const Int_Rep_And_Format_T * Layout);
```

The BigUSet (big integer unsigned set) routine shall set the low part of the big integer *Dest to NewValue, and shall set the other data bits (and sign bit, if any) to zero.

8.2.38 BigXOr

```
#include <bigint.h>
void BigXOr(
        Big_Int_T * Result,
        const Big_Int_T * A_Number,
        const Big_Int_T * Another_Number);
```

The BigXOr (big integer exclusive-or) routine shall set the big integer at *Result to the bitwise exclusive-or of the big integers *A_Number and *Another_Number. The sign bits (if any) are exclusive-ORed together, just as the corresponding data bits are, in a bitwise fashion.

8.2.39 BigZero

```
#include <bigint.h>
void BigZero(
        Big_Int_T * Result);
```

The BigZero (big integer zero) routine shall set the big integer *Result to all zero bits (including the sign bit, if any).