WG14/N318

X3J11/94-002

C-Packed.Txt

1.1 05-14-1992 drt.
1.2 12-18-1993 drt.

A Proposed Extension to the ANSI C Programming Language

by David R Tribble

May 1992
Revised Dec 1993

INTRODUCTION

With the need for support for commercial applications, and with the advent of newer and more powerful CPU architectures, the addition of a packed decimal data
type to the C language should be considered.

Packed decimal, also known as binary coded decimal (BCD), is a representation of numeric values in exact decimal form; that is, each value is composed of decimal digits rather than binary digits.

DATA REPRESENTATION

Internal representation of packed decimal values varies from CPU to CPU, but a typical representation (as found on VAX, IBM/370, Intel, and Motorola CPUs, and others) is:

- 4 bits (one nybble) per decimal digit
- two digits per (8-bit) byte
- one nybble containing a sign (positive/negative) indicator

The sign nybble typically has one or more values that indicate a positive sign and one or more that indicate a negative sign. Some representations allow for a sign nybble indicating no sign (or unsigned). For example, the value:

+1234567

is represented on a VAX-11 with the following bytes:

12 34 56 7C

where the digits `1' thru `7' are placed in the lowest address to the highest, the highest digit in the the high (left) nybble, with the rightmost low nybble holding the sign nybble. The sign nybble is allowed the following values:

A positive
B negative
C positive (preferred)
D negative (preferred)

E positive
F unsigned (preferred)

Other nybble values (0 thru 9) are illegal, but may be treated as unsigned or positive by the CPU. Some CPUs generate exceptions for invalid sign nybbles or invalid digit nybbles.

Some implementations reserve a single bit to indicate sign, rather than an entire nybble. Unused bits are ignored, or may be used to indicate special values (such as overflow, error, not-a-number, etc.). For example, the high bit of the high-order nybble could represent the sign, so that these values would be represented:

```
01 23 45 67   = +1234567
81 23 45 67   = -1234567
```

Note that some representations allow for both a positive and a negative zero, and some allow an unsigned zero as well:

```
00 00 00 0C   = +0
00 00 00 0D   = -0
00 00 00 0F   = 0 (unsigned)
```

Packed decimal values may be specified in three sizes, and as signed or unsigned. A `short' packed type should have at least 7 digits, and is typically representable in 32 bits. A `long' packed type should have at least 15 digits, and is typically representable in 64 bits. A `plain' packed type (which is specified as `packed' without a `short' or `long' preceding it) is not shorter than a `short' packed type and not longer than a `long' packed type, and may be identical to one of them. For example, one implementation may choose the following sizes:

short packed	7 digits, 32 bits
plain packed	7 digits, 32 bits (same as short packed)
long packed	15 digits, 64 bits

Another implementation may choose these sizes:

short packed	7 digits, 32 bits
plain packed	15 digits, 64 bits
long packed	23 digits, 96 bits

Signed and unsigned packed types are the same size, e.g., the signed short packed type is the same size as the unsigned short packed type. Depending upon the implementation, unsigned values may have either the same number of digits as or one more digit than signed values; a CPU may represent unsigned numbers with an even number of digits and no sign nybble, or may use a sign nybble with a special value to indicate unsigned, or may use a sign nybble but ignore its value. If an unsigned packed value has the same number of digits as a signed packed value, it will have half of the numerical range, since negative values are not included. If, on the other hand, an unsigned value has an extra digit, it has five times the range as a signed value, since it has an extra digit but no negative values.

The combinations of short, plain, and long types combined with signed, unsigned, and plain give the following possible packed decimal data types (some of which may be identical):

3 [illegible]

(plain) packed
(plain) short packed
(plain) long packed

signed packed
signed short packed or short signed packed
signed long packed or long signed packed

unsigned packed
unsigned short packed or short unsigned packed
unsigned long packed or long unsigned packed

LEXICAL EXTENSIONS

A new keyword, `packed', would be added. (Or, the word `decimal' could be used.)

Packed decimal constants would be a new class of lexical token. These would resemble normal integer constants with a `P' or `p' suffix. For example:

123P	signed (plain) packed
1234567p	signed (plain) packed
99999PL	signed long packed
99999PU	unsigned (plain) packed
99999PUL	unsigned long packed

SYNTAX EXTENSIONS

The grammar for the C language would be extended thus:

type-name:

PACKED
SHORT PACKED
LONG PACKED
SIGNED PACKED
SIGNED SHORT PACKED
SIGNED LONG PACKED
UNSIGNED PACKED
UNSIGNED SHORT PACKED
UNSIGNED LONG PACKED

SEMANTIC EXTENSIONS

Packed decimal values may be used within expressions, passed as arguments to functions, and returned from functions. Packed values may be r-values or l-values, and may be operands of the address-of operator (&).

It is unclear whether or not bitfields within structs and unions may be of type `packed'. What constitutes the shortest possible packed data value is also unclear. If unsigned packed bitfields are allowed, they could be restricted to being only multiples of 4 bits long. If signed packed bitfields are allowed, they could also be restricted to multiples of 4 bits with a minimum length of 8 bits to allow for a sign nybble.

Packed values within expressions are subject to data type promotions. The `usual numeric conversion' rules would be amended thus:

> ...
>
> If the operand is a short packed decimal value, it is converted to a plain packed decimal value.
>
> If the operand is an unsigned packed decimal value, it is converted to a signed packed value of the same size. (Note that this differs from the rules for binary integer types, since signed and unsigned packed numbers of the same size have the same number of possible positive values; it is not possible to get an overflow by converting an unsigned packed number into a positive signed packed number.)
>
> If one of the operands is a long packed decimal value and the other is a shorter packed decimal value, the shorter operand is converted to the longer type.
>
> If one operand is a packed decimal value and the other is binary integer, the operand with the type of lesser numeric range is converted to the type of the other operand. (E.g., if signed long int can hold more digits than signed packed, the packed value is converted into a long int value.)
>
> If one operand is a floating point operand and the other is a packed decimal value, the packed decimal value is converted to the type of the floating point operand.
>
> ...

This is a rough first cut, and needs to be refined to include rules for dealing with differences in signed and unsigned representations (i.e., if a signed packed value contains fewer digits that an unsigned packed value), as well as short, plain, and long representation differences. Essentially, the rules should make intuitive sense for realistic implementations. Default promotions for short packed types should be to convert them to plain packed types (such as for arguments to functions without prototypes) or to ints of the appropriate size (such as within arithmetic expressions). Expressions with both int and packed operands should result in efficient implicit conversions to the widest intermediate type, so as to preserve the arithmetic value, and if possible the sign, of the operands. On the other hand, the promotions should at the same time yield an intermediate type that is efficient for arithmetic computations. So, for example, while it might involve less steps to convert an int operand to a packed so that both operands of the addition operator can be packed, it may be more efficient in the long run to convert both operands to long int prior to adding them. On the other hand, it may be wiser to treat packed types as wider types than binary types, but less so than floating point types. Thus an expression with packed operands incurs the penalty of a wider type, just as an expression with floating point operands does.

Assignments and typecasts to and from packed decimal types are permitted. Any numeric type may be converted, although there may be some loss of precision or truncation of high-order digits. Some implementations may raise exceptions for certain conversions (such as attempting to convert a packed value with non-decimal digits into an int). Assignment of a packed value to a packed variable may `normalize' the value by correcting non-decimal digits and producing a `preferred' sign nybble; this is preferrable to, but more computationally expensive than, simply copying the value into the variable.

Some representations may make it possible for a short unsigned packed value to convert directly to a plain int without loss of precision, while at the same time a short signed packed value would also suffer some loss. Some representations may make it possible to convert a plain signed packed value into a long int without loss, while a plain unsigned packed value will not fit into an unsigned long int.

Packed decimal arithmetic involving two operands of the same size results in a value of the same type; i.e., a plain packed value added to a second plain packed values results in a plain packed sum, not a long packed sum. Overflows may raise exceptions in an implementation, or may result in the largest possible packed value with the appropriate sign, or may simply result in a value with truncated high order digits. The arithmetic operators *, /, %, +, -, unary +, and unary - have their usual semantic meanings.

Arithmetic operations on packed data should result in `normalized' or `preferred' representations, i.e., positive, negative, and unsigned values should have the `preferred' sign nybbles, and all digits should be valid decimal digits. Whether or not operations performed on invalid packed data values causes exceptions or default result values is implementation-defined. (Whether or not simple assignments should `normalize' packed values is unclear.)

The right shift and left shift operators (>> and <<) may be deemed illegal for packed decimal operands, or may be defined as resulting in a value that is shifted by a given number of decimal digits, by analogy to shifting a binary value by a certain number of binary digits. Thus an unsigned packed shift can be used as a fast multiply or divide by ten in the same way a binary shift can be used as a fast multiply or divide by two. Whether or not the sign is preserved, lost, or shifted is unclear.

Comparisons of packed decimal values should operate intuitively. Whether negative zero is less than positive zero is defined by the implementation; it is preferred that all zero values compare equal:

00 00 00 0C	positive zero
00 00 00 0D	negative zero
00 00 00 0F	unsigned zero

If positive and negative zero are considered different, a method for determining that a value is negative zero should exist, such as by a comparison to the constant 0. On the other hand, an argument can be made that both positive and negative zero should compare equal to 0, but not necessarily equal to each other. Perhaps the most compelling argument is that all zero values do compare equal, and that they also compare equal to all zero values of other types, such as int and pointer (NULL).

Note that implementations with multiple legal sign nybble values will consider some values as equal which do not have exactly the same bit pattern:

12 34 56 7C	+1234567, preferred sign
12 34 56 7A	+1234567, alternate sign

12 34 56 7F 1234567, unsigned

Comparisons involving packed values with invalid signs or digits is implementation-defined. Whether or not an exception is raised is implementation-defined.

The logical negation operator (!) retains the same meaning as for the other numeric types, being essentially a comparison to zero. This also applies to the ternary conditional operator (? :) and the logical binary operators (&& and | |).

The bitwise operators & (and), ^ (exclusive-or), and | (or) are illegal for operands of packed data types.

The bitwise complement operator (~) may be deemed illegal for packed operands, or may be defined as resulting in the nine's complement of the operand, by analogy to the one's complement of a binary operand. Whether the sign is ignored, lost, complemented, or normalized is unclear.

Since array indices are, by definition, of type unsigned int or unsigned long int (size_t), it is unclear as to whether packed expressions may be used as indices. If this is legal, then packed array index expressions are implicitly typecast to size_t values. However, if packed expressions are treated in a way similar to floating point expressions, then an explicit cast is required.

Pointer arithmetic involving packed operands is unclear, and is similar to the issue of array subscript expressions.

Packed initializer expressions are legal. However, is it unclear as to whether packed initializers for enum constants are legal; implicit typecasts to int or long int may be assumed, or explicit casts may be required.)

LIBRARY EXTENSIONS

The formatted input/output functions would be modified to allow for packed decimal types. Specifically, these functions would be enhanced:

printf()	scanf()
fprintf()	fscanf()
sprintf()	sscanf()
vprintf()	vscanf()
vfprintf()	vfscanf()
vsprintf()	vsscanf()

The `fmt' argument would be enhanced to allow for a `D' specification, indicating an argument of packed decimal type:

%(width)(.(min))(l | h)D

(Note that the letter `P' could be chosen instead.)

Like the %d and %u format specifications, the %D specifier takes an optional width and an optional minimum size. If %lD is specified, the argument is expected to be a long packed value; %hD specifies a short packed value (which is only valid for the scanf functions); %D specifies a plain packed value.

How to distinguish between a signed and an unsigned packed value is problematic; this will make a difference on implementations that use a different sign nybble to represent unsigned values. Using a `D' to indicate signed packed decimal and a `U' to indicate unsigned packed decimal is a possibility.

The following functions need to be added to the standard library, by analogy zto the atoi() and strtol() functions:

```
packed atop(const char *a)
packed strtop(const char *s, char **end, int radix)
long packed strtolp(const char *s, char **end, int radix)
unsigned packed strtoup(const char *s, char **end, int radix)
unsigned long packed strtoulp(const char *s, char **end, int radix)

packed pabs(packed x)
long packed lpabs(long packed x)
```

The following types and functions may also be added to the standard library, by analogy to the div() and ldiv() functions:

```
typedef ... pdiv_t;

pdiv_t pdiv(packed numer, packed denom)
lpdiv_t lpdiv(long packed numer, long packed denom)
```

The following functions or macros may also be added to the standard library, by analogy to the isnan() functions:

```
int pisnan(packed x)
int lpisnan(long packed x)
```

If an implementation chooses to raise an exception when operating on packed values containing invalid signs or digits (i.e., invalid bit patterns), then an appropriate signal should be added to <signal.h>:

SIGDEC   Invalid packed decimal value

CLOSING REMARKS

If packed decimal data types are accepted into ANSI C, the next step is to enhance the ANSI C++ definition as well.

Unix implementations, notably System VR4, BSD, Posix, and FIPS, will also need to benefit from this enhancement.

Not all of the issues discussed in this specification need to be addressed in the ANSI C definition; the whole business of signed and unsigned packed types may be deemed too complicated at the present time, for instance. The ANSI committee may also choose to make packed decimal data types an `optional' conformance item, leaving the decision to implement it up to the compiler writers; packed decimal types do not need to exist for all C compilers, after all (embedded microcontroller systems, for example).