Farance Inc.

subject: **Specification-Based Extended Integer Range, Revision 3 (5.1)**
document: **WG14/N459 X3J11/95-060**
file: **`c9x/extended-integers/sbeir.*`**

date: **1995-08-25**

from: **Frank Farance**
+1 212 486 4700
frank@farance.com

William Rugolsky, Jr.
+1 212 486 4700
rugolsky@farance.com

ABSTRACT

This proposal extends the C language notion of integral types to a wider range of hardware architectures, notably 64-bit and 128-bit applications. Many programmers have had problems porting existing 32-bit code to 64-bit systems. The heart of the problem lies in the limited number of integral types in C (`char`, `short`, `int` and `long`), and the assumptions programmers have made about these types. Our solution is to make *explicit* the three *implicit* features that programmers assume: precision (number of bits), exactness of precision (are *exactly N bits* required or *at least N bits*), and performance (optimize for time or space). Rather than forcing the programmer into the four traditional integer types (with varying performance and precision across architectures), this proposal improves portability by allowing programmers to specify *needs* (clarifying their intent) and the compiler vendor to provide the best implementation of those needs. An additional benefit is that the extension is easy to implement in many compilers. This proposal only concerns additional range — bit/byte ordering and alignment and data representation are outside the scope of this proposal.

CONTENTS

1. PROBLEM STATEMENT

The type model employed by the ANSI/ISO C Standard reflects the language's roots in systems programming. By minimally specifying the precision and semantics of the numeric types, the language has been efficiently mapped onto the native types provided by a wide variety of machine architectures.

1.1 Attributes of Integer Types

All applications make explicit and implicit assumptions about integral types. These assumptions are characterize in the following attributes.

- Signedness. Is the type `unsigned` or `signed`?
- Specified precision. The precision required, i.e., *N* bits.
- Exactness of precision. Whether the type provides exactly the specified precision (hereafter referred to as **exact semantics**), or at least that precision (hereafter referred to as **at-least semantics**).
- Actual precision. The available precision.
- Performance. Whether the representation is optimized for space or time.
- Range. The range of acceptable values.
- Non-participating bits. Any ''holes'' in the representation of the value.

In the context of this proposal, ''precision'' means the number of bits that participate in the value. For unsigned integers, an N-bit integer stores values in the range of 0 to ((2**N)-1). For signed integers, the range includes the values -((2**(N-1))-1) to ((2**(N-1))-1).

1.1.1 Exact Semantics

An **exact** type is one that has *exactly* N bits that participate in its value. The type may consume storage larger than N bits (e.g., to pad to a convenient byte or word boundary), but the remaining storage does not participate (holes) in the value. For example, a type of exactly 24 bits might be implemented as 3 8-bit bytes or as 32-bit word with the necessary bit mask operations.

In C terminology, an **exact** type of N bits is equivalent to a structure containing a bit field of N bits. Unlike structure bit fields, eact types may have the address-of operator (&) and the `sizeof` operator applied to the **exact** type objects.

1.1.2 At-Least Semantics

A type that has *at least* N bits that participate in its value. The type may consume storage larger than N bits for padding, but some storage might not participate in the value. For example, consider a system with 24-bit words and the double-word value has 47 bits that participate in value (48 bits of storage with a 1-bit hole). On this system, a type of at least 32 might be implemented as a 32-bit number with bit masks (*at least* 32 bits specified precision, 32 bits actual precision, and 16 non-participating bits). Another possible implementation is to use the 47-bit double-word value (*at least* 32 bits specified precision, 47 bits actual precision, and 1 non-participating bit).

1.1.3 Specified Precision

When specifying an integer type, the program specifies how much precision it *needs* in bits (*N*). The exactness attribute determines whether *N* is interpreted as an exact specification or a minimum specification.

1.1.4 Actual Precision

Although *N* bits may be specified for an **at-least** type, *M* may be available (*M* >= *N*) for use. For example, an integer type of at least 16 bits (**specified** precision) may be implemented as 32 bits (**actual** precision) on a 32-bit architecture.

For **exact** types, the **actual** precision is always the same as the **specified** precision.

1.2 Conformance Level and Style

Standard C has two levels of conformance with respect to programs running in a hosted environment.

- Strictly conforming program. The program ‘‘shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior, and shall not exceed any minimum implementation limit’’.
- Conforming program. ‘‘Conforming programs may depend upon nonportable features of a conforming implementation’’. The standard header `<limits.h>` is used to determine the maximum usable precision and range.

These two levels of conformance produce two styles of coding:

- Minimalist style. This coding style uses integer precision, range, operations, and values that are the same across all implementations. In other words, this style makes no assumptions about implementation limits.
- Adaptive style. This coding style adapts the program to the capabilities of the implementation.

For example, the program:

```
main()
{
        printf("%d\n",INT_MAX-INT_MAX);
        exit(0);
}
```

is a strictly conforming program: its output remains the same regardless of implementation. However, the program isn’t in a minimalist style because some of the values it uses (e.g., `INT_MAX`) are dependent upon the implementation limits.

NOTE: Minimalist style and adaptive style aren't terms in the C Standard. These styles can apply to programs or program fragments.

1.2.1 Exactness vs. Conformance Style

A common misconception is that minimalist programs use **exact** semantics, while adaptive programs use **at-least** semantics. Exactness is a separate concept from conformance style. The following examples show the distinction.

Example #1: Minimalist Style and Exact Type

A minimalist style program can use an **exact** type to get precise arithmetic semantics, e.g., bit shift for calculating a CRC checksum. In Standard C, an `unsigned` bit field, could be used for this purpose. Standard C has the limitations:

- Only up to 16 bits can be used portably.
- There are no pointers to bit fields.
- Performance attributes cannot be specified. `register` is not possible in a `struct` (optimizing for time). The `pack` pragma is not standardized (optimizing for space).

Example #2: Minimalist Style and At-Least Type

A minimalist style program can use an **at-least** type when:

- Portable bit-shift operations are not required. Bit shifting `unsigned` integers can produce different results on machines with different precision, e.g., `(0x7FFF<<2)>>2`. The Standard C integral types, `char`, `short`, `int`, `long`, may be used as long as their minimum precision (8, 16, 16, 32 bits) is not exceeded. Most programs assume that of the 16-bit types, `short` is the smaller of the two and `int` is the faster of the two.
- Portable bit-shift operations and more than 16 (but not more than 32) bits of precision are required. A minimalist style program must use an `unsigned long` with each operation bracketed bit masks to clear any excess bits. This simulates an **exact** type for up to 32 bits.
- The address-of `&` operator must be applied to the value. In Standard C, there are no pointers to **exact** types (i.e., bit fields), so an **at-least** type must be used.

There are no minimalist style programs (or strictly conforming programs) that use more than 32 bits (**exact** or **at-least**) precision.

Example #3: Adaptive Style and Exact Type

Using right bit-shift operations on a bit-field.

Example #4: Adaptive Style and At-Least Type

Using any extra available precision outside the minimum specified precision, e.g., using more than 16 bits of an `int` on systems that have 32-bit `int`'s.

Using right bit-shift operations on any signed integer type.

1.2.2 Extending Minimalist Style Programs

The following features are required for extending precision of minimalist style programs.

- Providing **exact** types with precision greater than 16 bits.
- Providing pointer semantics (i.e., applying the address-of `&` operator) to **exact** types.
- Providing **at-least** types with precision greater than 32 bits.
- Providing a simpler paradigm for selecting the correct type (see below).
- Providing support for `printf`, `scanf`, and `strtod` families of functions.
- Providing a consistent, easy-to-understand promotion rules with respect to **specified** precision.

1.2.3 Extending Adaptive Style Programs

The following features are required for extending precision for adaptive style programs.

- Providing performance attributes (`fast` and `small`).
- Providing `MIN` and `MAX` macros for arbitrary types.
- Providing a consistent, easy-to-understand promotion rules with respect to **actual** precision.

1.3 Determining The Correct Integer Type

With any type system, the programmer must make a choice of which type to use. The following information must be known prior to choosing a type:

- Specified precision — *N* bits.
- Actual precision of the available types (e.g., `char`, `short`, `int`, `long`) in the implementation.
- Exactness of precision — **exact** or **at-least** semantics.
- Performance attributes — optimized for space, time, or neither.
- Conformance style. Will the program use the type in a minimalist or an adaptive style?
- Addressability. Must the program create a pointer to the type?

The algorithm for determining the correct type can be:

- Known by the programmer. In this case, the programmer chooses a type that is expected to be portable (minimalist or adaptive, as desired) across all implementations. The programmer then ‘‘hard codes’’ the type name into the program.
- Embedded in the preprocessor directives in the program. Here, preprocessor directives inquire, examine, and determine the nature of some of the basic C types.
- Determined in a separate pass. A tiny, experiment program is compiled and run to determine the appropriate types for the application program. The experiment program generates a header which defines the appropriate types for the application.

There are several disadvantages to these methods. Rarely do programmers document their intent within the code.

```
/*
 * Code for some 32-bit implementation.
 *
 * Programmers RARELY document the intention of the type.
 * Programmers, more often, document the purpose of the
 * variable.
 */
int a; /* signed fastest int at least 16 bits */
short b; /* signed smallest int at least 16 bit */
int c; /* signed int at least 16 bits */
int d; /* signed int exactly 16 bits */
int e; /* signed fastest int at least 32 bits */
int f; /* signed int exactly 32 bits */
```

In fact, most programmers believe that the type they chose *documents itself*. Since there are roughly 23 scenarios (see below) in Standard C than map into 5 ‘‘types’’ (`char`, `short`, `int`, `long`, and bit fields), **it is impossible that the ‘‘type’’ documents the intent of the programmer**.

Even good programmers that document intent still have problems because they must map their intent (precision, conformance style, performance, addressability) — hundreds of possibilities — into the 5 ‘‘types’’. The mapping of type intents to the 5 ‘‘types’’ varies from machine to machine. Even good programmers will have to make manual changes as they port code from machine to machine.

The preprocessor approach can become especially complex for conforming programs if any performance attributes (e.g., smallest storage) are required to be determined at preprocessor time. The preprocessor cannot inquire about any type (e.g., its size) defined with `typedef` because type definition is performed at a later phase in program translation.

Because the preprocessor translation phase is before type definition, some programmers solve the problem by writing an experiment program (see above). However, it is not possible run an experiment program when the execution environment is different from the translation environment.

Regardless, any of these methods to determine the correct type can be described by the following algorithms. These algorithms demonstrate what is performed by the programmer, the preprocessor, or an experiment program.

1.3.1 Algorithm for Standard C

The following algorithm is used for Standard C to determine the correct type for a program. The type-determination parameters (specified precision, actual precision of available types, exactness, performance, conformance, addressability — see above) are the inputs to this algorithm. The algorithm emits one of five ‘‘types’’ (`char`, `short`, `int`, `long`, and bit fields) as the correct ‘‘type’’. To simplify the presentation of this algorithm, the signedness is not included. The programmer would prepend `unsigned` or `signed` as appropriate.

- If the program is strictly conforming, then:
  - — If an **exact** type is required:
    - A. If the **specified** precision is <= 16, then:
      - a. If address-of `&` operator will be used:
        - **Scenario #1.** Use `unsigned int` with bit masks.
      - b. Otherwise:
        - **Scenario #2.** Use `unsigned` bit field.
    - B. Otherwise, if the **specified** precision is <= 32, then:
      - **Scenario #3.** Use `unsigned long` with bit masks.
    - C. Otherwise:
      - Not possible directly with C types.

— Otherwise, it must be an **at-least** type:

   A. If the **specified** precision is <= 8, then:
      - **Scenario #4.** Use some `char` type.

   B. Otherwise, if the **specified** precision is <= 16, then:
      a. If optimizing for space (smallest storage), then:
         - **Scenario #5.** Use some `short` type.

      b. Otherwise, if optimizing for time (fastest operations), then:
         - **Scenario #6.** Use some `int` type.

      c. Otherwise, if optimizing for space and time:
         - Not possible with C types.

      d. Otherwise:
         - **Scenario #7.** Use some `int` type.

   C. Otherwise, if the **specified** precision is <= 32, then:
      - **Scenario #8.** Use some `long` type.

   D. Otherwise:
      - Not possible directly with C types.

- Otherwise, the program must be conforming:

— If an exact type is required, then:

   A. If the **specified** precision == the **actual** precision of `char`, then:
      - **Scenario #9.** Use some `char` type.

   B. If the **specified** precision == the **actual** precision of `short`, then:
      - **Scenario #10.** Use some `short` type.

   C. If the **specified** precision == the **actual** precision of `int`, then:
      - **Scenario #11.** Use some `int` type.

   D. If the **specified** precision == the **actual** precision of `long`, then:
      - **Scenario #12.** Use some `long` type.

   E. If the **specified** precision <= the **actual** precision of `int`, then:
      a. If address-of `&` operator will be used:
         - **Scenario #13.** Use `unsigned int` with bit masks.

      b. Otherwise:

- **Scenario #14.** Use `unsigned int` bit field.

F. If the **specified** precision <= the **actual** precision of `long`, then:

- **Scenario #15.** Use `unsigned long` with bit masks.

G. Otherwise:

- Not possible directly with C types.

— Otherwise, it must be an **at-least** type:

A. If optimizing for space (smallest storage), then:

a. If the **specified** precision <= the **actual** precision of `char`, then:

- **Scenario #16.** Use some `char` type.

b. Otherwise, if the **specified** precision <= the **actual** precision of `short`, then:

- **Scenario #17.** Use some `short` type.

c. Otherwise, if the **specified** precision <= the **actual** precision of `int`, then:

- **Scenario #18.** Use some `int` type.

d. Otherwise, if the **specified** precision <= the **actual** precision of `long`, then:

- **Scenario #19.** Use some `long` type.

e. Otherwise:

- Not possible directly with C types.

B. If optimizing for time (fastest operations), then:

a. If the **specified** precision <= the **actual** precision of `int`, then:

- **Scenario #20.** Use some `int` type.

b. Otherwise, if the **specified** precision <= the **actual** precision of `long`, then:

- **Scenario #21.** Use some `long` type.

c. Otherwise:

- Not possible directly with C types.

C. If optimizing for space and time:

a. Not possible with C types.

D. Otherwise:

a. If the **specified** precision <= the **actual** precision of `int`, then:

- **Scenario #22.** Use some `int` type.

b. Otherwise, if the **specified** precision <= the **actual** precision of `long`, then:

- **Scenario #23.** Use some `long` type.

c. Otherwise:

- Not possible directly with C types.

1.3.2 Algorithm for Kwan Proposal

The algorithm for the Kwan Extended Integer Range proposal is basically the same if 64-bit types are excluded. The Kwan types `int16_t` and `int32_t` would replace `int` and `long`, in the **exact** portion of the algorithm. The types `int_least_8_t`, `int_least_16_t`, `int_least_32_t`, would replace `char`, `short`, `int`, `long`, in the **at-least** portion of the algorithm.

The Kwan proposal improves Standard C be eliminating only one scenario (#18). This leaves 22 scenarios than map into 7 ‘‘types’’ (`int8_t`, `int16_t`, `int32_t`, `int_least_8_t`, `int_least_16_t`, `int_least_32_t`, and bit fields). If 64-bit types are included, this adds 7 more scenarios for a total of roughly 29.

In summary, the Kwan proposal doesn’t change much of the methods for determining the correct type. Still, the programmer, the preprocessor, or an experiment program must execute this algorithm. With 64-bit types, there are roughly 29 scenarios that map into 9 ‘‘types’’ (`int8_t`, `int16_t`, `int32_t`, `int64_t`, `int_least_8_t`, `int_least_16_t`, `int_least_32_t`, `int_least_64_t`, and bit fields). Regardless, there is still a type-determination algorithm that needs to be executed that loses information.

1.3.3 Algorithm For This Proposal

The following is the type-determination algorithm an application would use with this proposal. Note that this algorithm is *significantly* simpler than the algorithm for Standard C or the Kwan proposal, and, **the algorithm loses no information**.

A. Determine signedness of desired type.

1. If the type is signed, then:

   - Add `signed`.

2. Otherwise:

   - Add `unsigned`.

B. Determine performance attributes.

1. If optimizing for space (smallest storage), then:

  - Add `small int`.

2. Otherwise, if optimizing for time (fastest operations), then:
   - Add `fast int`.
3. Otherwise, if optimizing for space and time, then:
   - Not possible.
4. Otherwise:
   - Add `int`.

C. Determine exactness of precision.

1. If an exact type is required, then:
   - Add `exact`.
2. Otherwise, an at-least type is required:
   - Add `atleast`.
3. Determine the precision required. Add `:`*N* where *N* is the number of bits required.

The following examples demonstrate the use this approach by defining types that are similar to standard C types.

```
typedef unsigned small int atleast:8 std_uchar;
typedef signed int atleast:16 std_short;
typedef signed fast int atleast:16 std_int;
typedef signed int atleast:32 std_long;
```

NOTE: This proposal **does not** require this definition for Standard C types — this is a sample mapping.

1.4 The Problem To Fix

The main problem to fix is not to lose information due to the mapping of type intents which are mapped into scenarios and finally mapped into ‘‘types’’. Another way of stating this is to match the implementation to what programmer wants.

1.4.1 Porting Problems

The primary motivation for Extended Integer Range is the porting cost (extra cost) and/or the lack of portable integral types (missing functionality) all while maintain good performance. Porting problems have existed since the beginning of C. The main difficulty is determining whether the code is written (or has been written) to minimum specifications (minimalist style programs) or to take maximum advantage of the machine (adaptive style programs).

The following is a brief history of porting problems.

1.4.1.1 16-Bit vs. 32-Bit Systems

The problems that are occurring now, porting 32-bit code to 64-bit systems, are similar to the problems of the late 1970's to early 1980's when porting code from 16-bit to 32-bit systems. During that time, there was a large body code that assumed an `int` was exactly 16 bits. With the UNIX V7 C compiler (for the PDP-11, a 16-bit machine), the solution was to add the `short` and `long` types. This was consistent with the UNIX V32 C compiler (for the VAX, a 32-bit machine) that shortly followed.

16-Bit Culture

There was a large body of code that assumed an `int` was 16 bits. There were many other assumptions (e.g., the size of a pointer is the same size as an `int`), but they are not significant to this discussion.

The 32-Bit Porting Problem

Through the early to mid 1980's, the 32-bit porting issues were: usage of `int` when other types were more appropriate, and, assumptions that `int` would store a pointer (not true on 16-bit machines with ''far'' pointers). For each porting issue that was related to precision, the usage was analyzed to determine the original programmer intent:

- `int` is *exactly a 16-bit* type. In the porting effort, this would have been changed to `short`. `short` is not *exactly* 16 bits, but all the 16-bit and 32-bit machines at the time implemented `short` as 16 bits, so programmers assumed this to be true.
- `int` has *at least 16 bits* precision, but there were no performance requirements. This usage was left unchanged.
- `int` has *at least 16 bits* precision, but the usage was optimized for speed. This usage was left unchanged since `int` was the *fastest type of at least 16 bits* on both 16-bit and 32-bit machines.
- `int` has *at least 16 bits* precision, the usage was optimized for (minimum) storage. In the porting effort, this would have been changed to `short` because either assumption: `short` is exactly 16 bits (wrong), or `short` is smaller than `int` (not exactly right; correct: `int` is not smaller than `short`).
- `int` has *exactly 32 bits* precision. This problem existed in porting Berkeley UNIX code (developed on a 32-bit VAX) to 16-bit machines. In the porting effort, this would have been changed to `long`, but this is still incorrect. `long` has *at least 32 bits* precision, not *exactly 32 bits* precision. Since most of the machines at the time implemented `long` as 32 bits, this was not a problem then.
- `int` has *at least 32 bits* precision. As above, this problem existed in Berkeley UNIX. In the porting effort, this would have been changed to `long`.

Standards Effort

K&R C first clarified an acceptable coding paradigm. From an early version of K&R C:

> ‘‘The intent is that `short` and `long` should provide different lengths of integers where practical; `int` will normally reflect the most ‘‘natural’’ size for a particular machine. ... each compiler is free to interpret `short` and `long` as appropriate for its own hardware. About all you should count on is that `short` is no longer than `long`.’’

The ANSI C Standard clarified this even further: `short` and `int` are *at least* 16 bits and `long` is *at least* 32 bits.

1.4.1.2 32-Bit vs. 64-Bit Systems

Difficulties encountered when porting code from 32-bit to 64-bit systems are due to implicit assumptions about one or more of these attributes for the integral types `char`, `short`, `int`, and `long`.

32-Bit Culture

The programmers of today assume that an `int` is 32 bits (most of the time) and a `long` is exactly 32 bits. Because an `int` is still used as a counter or array index on large objects (>32767 bytes or indexes), programmers assume that an `int` is now 32 bits. This confuses the availability of larger address spaces with (incorrect) assumption that `int` must be at least 32 bits.

Like the programmers of 15 years ago, today’s programmers haven’t been burned too much making certain (incorrect) assumptions: `short` is exactly 16 bits, `int` is 32 bits (occasionally 16 bits), `long` is exactly 32 bits. In fact, when programmers run into problems, e.g., a machine with a 16-bit `int`, the programmers attribute the problem to ‘‘old code running on old machines’’ (implying the code was defective), rather than the programmer was making incorrect assumptions.

The 64-Bit Porting Problem

When porting code to 64-bit machines, the problem has all the same problems of 10 years ago: take the list of 32-bit porting problems (above) and replace ‘‘16 bits’’ and ‘‘32 bits’’ with ‘‘32 bits’’ and ‘‘64 bits’’, respectively. Additionally, the code that runs on 64-bit machines still has to work on 16-bit machines, which causes more porting problems.

Another problem with 64-bit machines is that the mapping of the four basic integral types (`char`, `short`, `int`, `long`) varies from machine to machine. 16-bit implementations generally mapped the types into 8, 16, 16, and 32 bits. 32-bit implementations generally mapped the types into 8, 16, 32, and 32 bits. However, several mappings are possible (and reasonable) on 64-bit machines. The following mappings are typical:

```
char    short   int     long
8       16      32      32
8       16      32      64
8       16      64      64
8       32      32      64
8       32      64      64
8       64      64      64
```

For programmers writing in a minimalist style, they could continue to use a mapping of 8, 16, 16, and 32 bits. For programmers that write in an adaptive style (i.e., take advantage of the maximum available precision), they will need to resort to complicated preprocessor tricks or experiment programs and, possibly, `<limits.h>`.

Standards Effort

With the development and use of 64-bit machines, there has been much interest in the standardizing integral types on these machines. It is not possible to standardize on a specific mapping of the four basic integral types (`char`, `short`, `int`, `long`) to native types on a particular machine. This is because programmers have made (different) assumptions all along about the use of these types. Each mapping of the four types has advantages and disadvantages. Some of those disadvantages are that existing code breaks and needs to be fixed (‘‘ported’’).

Another way of stating this is that there was no agreement on the mapping of type intents to C ‘‘types’’. Of course, there was no agreement on this mapping because the advantages and disadvantages aren't the same for each vendor.

Other standards efforts have investigated a new type `long long`, but this makes the problem *worse* (see below).

1.4.1.3 Solving The Same Problem Again

Even if it were possible to choose an appropriate mapping or standardize on a new type, this would create a whole new set of problems: as programmers adapted and felt comfortable with the popularity of 64-bit machines (5-15 years from now), they would still be faced with a portability problem again (going to 128-bit applications), yet much more complex due to growth of assumptions (not documented or not accessible in programs) and limited knowledge about the use of a type (loss of information).

Solving portability problems (i.e., investigating the use and intention of each declaration) in the future will only get worse since there will be more code to port (investigate).

The solution is to reduce assumptions and make obvious the intent of the usage. Not only will programmers be able to understand this, but ‘‘preening’’ programs (e.g., `lint`) will be able to discover portability problems.

1.4.2 Loss of Information

The loss (or lack) of information when determining a type is the primary cause of porting problems. Similarly, the implementor must account for the loss of information and choose a mapping of the four basic integral types that meets the perceived needs of the programmer.

The porting problems are usually ‘‘after the fact’’, i.e., resolving problems porting code from 32-bit systems to 64-bit systems. There are still problems ‘‘before the fact’’. When a programmer writes new code, he/she still must determine an appropriate type. Thus, new problems are being created today due to a lack of capability (or a lack of understanding).

This section demonstrates that the loss of information is inevitable if the programmer is not allowed to specify his/her needs.

1.4.2.1 Standard C

The algorithm above demonstrated that Standard C *requires* roughly 23 scenarios, yet provides only five ‘‘types’’ for implementation. One suggested solution is to provide a header for each scenario. There are two problems with this. The first problem is that because `typedef` is interpreted in a translation phase later than `#include` files, it may be impossible to determine correct types based upon preprocessor logic. The second problem is that the **exact** types in the sample algorithm were useful on 8-bit byte machines, but not on word-oriented machines. The header could include an **exact** type for every bit precision up to 128 bits, but given the combinations required (`unsigned` vs. `signed`, `fast` vs. `small` vs. unoptimized), this would require about 1000 type definitions.

Another possibility is to train programmer to document the usage at the point of declaration. While this may help, the compiler won’t be able to detect any portability problems. Even with appropriate program documentation, porting is still a manual task because the mapping of type intents to C ‘‘types’’ varies among implementations.

1.4.2.2 Kwan Proposal

Although the Kwan proposal provided for more mappings (29 scenarios map to 9 types), it doesn’t solve the problem with losing information. Thus, it is not expected that the Kwan proposal will significantly reduce portability problems.

1.4.2.3 Other Proposals

Other proposals to provide Extended Integer Range, for example `long long`, have assumed that the central problem is the need for an additional type. At first glance, this may appear to help, but when comparing this to Standard C, the `long long` proposal produces *fewer* mappings per scenario (20%) than Standard C (22%). So `long long` makes the problem *worse* because more information is lost.

1.4.3 Minimizing Preprocessor Tricks

Using the preprocessor to determine the correct type can be complex and faulty. In some cases it may be impossible, using the preprocessor alone, to determine the correct type. These preprocessor ‘‘tricks’’ shouldn’t be the paradigm for determining the correct type.

1.4.4 Limited Precision

For strictly conforming programs, Standard C provides a maximum precision of 16 bits for **exact** types and 32 bits for **at-least** types. The minimum precision for both should be at least 128 bits to anticipate the needs of the near future.

1.4.5 Class Libraries Are Impractical

One approach is to implement any new types as a class library in C++. Given the lack of promotion rules for classes, the class library developer would need to be specific about promotion rules by writing a prototype of each arithmetic operator with each combination of operands. Even if only 8-, 16-, and 32-bit types were provided, this would require several thousand prototypes in scope. Of course, too, there would need to be code written for each prototype. This would be impractical for most systems.

1.5 Summary

Programmers make assumptions about the following qualities of integral types:

- The specificed precision (for strictly conforming programs).
- The available precision (for conforming programs).
- Bit-shift operations — are they portable?
- Exact (mask extra bits) vs. at-least.
- Overhead of masking bits.
- Pointer operations — is the address-of operator `&` allowed?
- Storage optimization — is this the smallest type?
- Speed optimization — is this the fastest type?

The porting problems of the past and present have been caused by:

- The loss of information when choosing a type. Using Standard C, there are roughly 23 scenarios that map onto five ‘‘types’’ (`char`, `short`, `int`, `long`, and bit fields).
- Excessive effort required to determine the intent of the programmer. *Each* variable must be analyzed when porting the system to a new architecture.
- The intent of the original programmer can be blurred by later maintenance on the program. For example, the original programmer needed *at least 16 bits*, so `int` was used. After the system had been ported to a 32-bit system, a second

programmer (seeing `int` on a 32-bit system) interprets the program having *exactly 32 bits*. A third programmer, when porting to a 64-bit system, changes the type to `short`, possibly optimizing for space, and knowing (incorrectly via the second programmer) that the code couldn't have used more than 16-bits because the type was `int`.

Adding the type `long long` makes the problem *worse*, not better. Standard C has 23 scenarios that map into 5 ''types'' — a mapping of about 22%. Assuming `long long` would provide *at least 64 bits* precision, this would imply roughly 30 scenarios that map into 6 ''types'' — a mapping of about 20%. The use of `long long` will cause the loss of *more* information, thus, it will create *more* portability problems.

Providing a header of 8-, 16-, 32-, and 64-bit types (e.g., the Kwan proposal) doesn't solve the problem either. While there may be a higher percentage of mapping scenarios to types, still, information is lost. Also, because `typedef` is interpreted in a later translation phase than headers, it may be impossible to use the preprocessor to determine the correct type.

Programmers don't document their intent when declaring a type. Programmers usually document the purpose of the variable they are declaring.

Implementing a type system like this as a C++ class library would be impractical because several thousand prototypes would be required.

This proposal provides mechanisms for specifying the minimum precision, the exactness of precision, and the performance, while leaving to the translator the choice of the actual precision. This division of attributes between the programmer and the translator allows the programmer to get closer to the hardware for maximum performance without sacrificing portability.

The following are problems not addressed by this proposal:

- Bit/byte ordering and alignment.
- Data representation.
- Language independent arithmetic.
- Language independent data types.

2. TYPE SYSTEM

This proposal achieves three goals:

1. It provides a simpler method of type-determination: the programmer just specifies what is needed. There is no complex algorithm, no preprocessor tricks, no experiment programs.
2. No information is lost when specifying the type because the programmer declares the intent of usage.
3. It provides more than 32 bits of precision, portably.

2.1 Conceptual Model

To motivate the proposed changes to the type model of Standard C, it is instructive to consider the methods by which programmers select types.

There are in principle three distinct types associated with an application programming interface: the application type, the interface type, and the implementation type. Ideally, all three are type-compatible (through promotion or demotion) and thereby avoid the need for explicit conversion. The application type is the type used directly by an application. The interface type is the type appearing in a prototype for the interface; it is a form recognizable to both sides of the interface. The implementation type is the type used to implement the interface functionality.

Though it is often the case that the three types are identical, this is not required. The purpose in distinguishing the various types is to tune the performance on each side of the interface. Often one side of the interface requires the fastest type for performing computations, while the other side of the implementation requires the most compact type for storage. For example, applications may (and often do) perform calculations on an integral value using the `int` type for performance reasons, even though the implementation of an API may eventually store the value in a `short`.

The method currently used for specifying portable interfaces is to hide the choice of integral type behind a `typedef` in a header. The mapping between the application types and the basic integral types is then accomplished by either editing the header by hand when porting to a new system, or providing preprocessor directives to select the appropriate types based upon the manifest constants defined in `<limits.h>`. As demonstrated above, either method is cumbersome, especially if the implementation type differs from the application type. Furthermore, the API implementer must make assumptions about the applications requirement's, such as whether the application needs to optimize the time or space associated with the type. These choices are, generally, outside of the scope of the API.

It is more straightforward and portable to define directly the required semantics for a type and let the translator choose the implementations. This approach has several immediate benefits:

- There is no need for elaborate preprocessor magic hidden in headers.

- Under certain circumstances, the application programmer can optimize the definition of a compatible application data type for time or space depending upon the application requirements.
- The portability of the implementation can be improved by specifying the semantics of the implementation type.

2.2 Proposed Semantics

2.2.1 Type Attributes

An integral type in the proposed type model has the following type attributes:

- Signedness. An integral type may be `signed` or `unsigned`.
- Specified precision. An integral **at-least** type has a specified minimum number of bits of precision (`:`*N*). For **exact** types, the the **specified** precision equals the **actual** precision.
- Exactness. An integral type may require either **exactly** (`exact`) specified precision, or **at-least** `atleast`) the specified precision.
- Performance. An integral type may be optimized for time (`fast`), optimized for space (`small`), or unoptimized.
- Actual Precision. The actual precision is determined by the translator and is an implementation-defined value not less than the specified precision. The value chosen by the translator represents the best available match to the other three attributes.

The mapping of type specifications to representations must satisfy the following axioms:

- The actual precision of two types differing otherwise only in their signedness attribute must be the same.
- The actual precision of types differing otherwise only in their performance attribute must be such that the actual precision of a type optimized for space is no greater than the actual precision of an unoptimized type, and the actual precision of an unoptimized type is no greater than the actual precision of a type optimized for time. In other words, the ordering from smallest to largest actual precision is: `small` <= unoptimized <= `fast`.
- The actual precision of two unoptimized types differing otherwise only in their specified precision must be such that the type with greater specified precision has actual precision no less than than the actual precision of the other type. In other words, if *specified(N) >= specified(M)*, then *actual(N) >= actual(M)*.

These axioms ensure that the promotion rules are consistent and sensible. The are several subtle points, though, that are worth emphasizing:

- The actual precision of two `small` types can be stated, if *specified(N) >= specified(M)*, then *actual(N) >= actual(M)*. Although this isn't stated in the axioms above, it can be deduced — a simple proof by contradiction.

- It is **not** true for two `fast` types that if *specified(N) >= specified(M)*, then *actual(N) >= actual(M)*. For example, the fastest 16-bit type might be 64-bits, while the fastest 32-bit type is 32 bits. The 64-bit value might be slower for 32-bit operations (e.g., multiplication) than a 32-bit value for 32-bit operations. Another way of stating this is that the ‘‘fastness’’ is only guaranteed for values that don’t exceed the **specified** precision. At first this may seem awkward, but it allows the programmer to ‘‘get close to the hardware’’ by achieving the performance objectives with the **specified** precision.

2.2.2 Promotion Rules

The **integral promotions** and **usual arithmetic conversions** are backward-compatible with the Standard. The integral promotions need only be modified to include the new types:

> In an expression, if an `int` can represent all values of the original integral type or bit-field, the value is converted to an `int`. Otherwise, if an `unsigned int` can represent all values of the original type, the value is converted to an `unsigned int`. Otherwise the type is unchanged.

Whereas the Standard currently refers to the **size** of an integer in subclause 6.2.1.2, **size** needs to be replaced by **actual precision**.

The purpose of the **usual arithmetic conversions** is to cause conversions that yield a common type for the binary operators requiring operands of the same type. The following summarizes the effect of the usual arithmetic conversions on the operands:

- Minimum Precision. All operands whose actual precision is smaller than the actual precision of `int` are promoted to `int`.
- Actual Precision. The actual precision of the result is at least the maximum of the actual precisions of the operands.
- Exact Semantics. If both operands are **exact**, the result is **exact**. Otherwise the result is **at-least**.
- Specified Precision. The specified precision of the result must be at least the maximum actual precision of the operands.
- Signedness. If both operands are signed, the result is signed. If bother operands are unsigned, the result is unsigned. Otherwise, if the operands differ in actual precision, the result is the the signedness of the operand with the larger actual precision. Otherwise, the result is unsigned.
- Performance. Performance attributes do not propagate into expressions.

2.2.3 Existing C Types

The existing C types, `char`, `short`, `int`, `long`, and their `unsigned` counterparts all map to an implementation-defined set of specification-based integral types. This means that an implementor is free to choose an appropriate mapping of these types, as long as the minimum precision specification (8, 16, 16, 32 bits) is met and the axiom `char` <= `short` <= `int` <= `long` must still hold true for both the **specified** and **actual** precision. For example, the following might be valid mappings:

```
/*
 * The following mappings are written as
 * "typedef"s to simplify the presentation.
 * An actual implementation would implement
 * the types directly rather than in a
 * header.
 */



/*
 * Minimal mappings defined by Standard C.
 * Note: "char" can be "signed" or
 * "unsigned".
 */
typedef unsigned int exact:8 std_char;
typedef signed int exact:16 std_short;
typedef signed int exact:16 std_int;
typedef signed int exact:32 std_long;


/*
 * Possible mappings for a 16-bit machine.
 */
typedef unsigned int exact:8 std_char;
typedef signed int exact:16 std_short;
typedef signed int exact:16 std_int;
typedef signed int exact:32 std_long;
```

```
/*
 * Possible mappings for a 32-bit machine.
 */
typedef unsigned int exact:8 std_char;
typedef signed int exact:16 std_short;
typedef signed int exact:32 std_int;
typedef signed int exact:32 std_long;
```

```
/*
 * Possible mappings for a 64-bit machine.
 * NOTE: There are many possible mappings.
 */
typedef unsigned int exact:8 char;
typedef signed int exact:16 short;
typedef signed int exact:32 int;
typedef signed int exact:64 long;
```

Note that these mappings are necessary: (1) so an `int` is defined (needed to define the minimum precision of calculations), (2) so existing code doesn't break. Once the implementor has defined the mapping of C types to specification-based types, the implementor must define the mapping of specification-based types to native machine types.

The reason for two levels of mapping is that the first level (mapping C types to specification-based types) binds the ''assumptions'' to a single set of specification-based types. This is necessary for traditional C types to interact with specification-based types. The second level of mapping is required to determine the **actual** precision and storage size of all specification-based types.

In the 64-bit example above, an `int` is defined as *at least 32 bits*. The **specified** precision exceeds the required precision (16 bits) in Standard C, so this definition is acceptable. However, a type of *at least 32 bits* may have 64 bits of **actual** precision. The reader may ask, ''why don't you just define it as *at least 16 bits* or *exactly 64 bits* if it will map to 64 bits **actual** precision anyway?''. There is no right answer here — as long as the minimum precision exceeds Standard C and the four basic types are ordered properly. An implementor that is providing a new 64-bit architecture with a large existing customer base of 32-bit code might choose the **specified** precision of *at least 32 bits* because it is based upon the **actual** precision of `int` in the code based the implementor and programmers are coming from.

Other 64-bit implementors going from 32-bit code may choose a mapping that causes the fewest portability issues, i.e., it maps closely to the assumptions held on the 32-bit code. These assumptions might be best mapped as:

```
/*
 * Possible mapping for a 64-bit machine
 * when programmers believed they only
 * ran on 32-bit machines.  Note there
 * is no 64-bit type.
 */
typedef unsigned small int exact:8 char;
typedef signed small int exact:16 short;
typedef signed small int exact:32 int;
typedef signed small int exact:32 long;
```

In this example, programmers do not have access to a 64-bit type *via the normal C types*. In fact, all the old 32-bit code should behave exactly the same on the 64-bit machine (not bad for portability!). The only problems may be with performance: the `int` as a 32-bit type might not be the fastest type. In these cases, the those `int` declarations can be changed to `signed fast int atleast:32`, assuming that the program won't use more than 32 bits (there are other considerations such as compatible arithmetic semantics for the larger type). Any new code that needs 64 bits precision won't work on the old 32-bit system (unless specification-based types are added), so new code can use, for example, `signed int atleast:64`.

Existing systems that support `long long` may consider mapping the type into a specification-based type, such as:

```
/*
 * This is not really a "typedef"
 * but code used for explanation.
 */
typedef signed int exact:64 long long;
```

2.3 Proposed Syntax

The proposed syntax is intended to expose the full type model to the programmer while introducing minimal changes to the the existing language syntax. The precise spelling of the keywords and macros is not central to the proposal and is left to the discretion of the Committee.

The proposal adds four new type specifiers: `atleast`, `exact`, `fast` and `small`. The type specifiers `atleast` and `exact` specify the exactness semantics and are mutually exclusive. The specifiers `fast` and `small` specify the performance characteristic, and are mutually exclusive.

The specified precision of a type is denoted by a colon `:` and an integral constant expression following the type specifier `atleast` or `exact`. For example, `int exact:32`. This usage does not conflict with the use of the colon `:` for specifying bit-fields. In the case of an anonymous bit-field there is a unique valid interpretation.

This proposal requires a change to the handling of integral constant expressions. The translator is required to evaluate constants with an implementation-defined precision of not less than 128-bit precision. (This minimum limit is compatible with the IEEE 1596.5 Standard for extended integer range.)

This proposal adds support for constants with programmer-specified precision. The notation is similar to the exponent notation for floating constants: the `P`*nn* suffix specifies an integer constant of *nn* bits. For example, `123P16` specifies a signed, 16-bit constant; `456P32U` specifies an unsigned, 32-bit constant.

3. SUPPORTING SERVICES

The supporting services provide methods for querying the attributes of a type. They are used by the formatted input/output services of the Standard Library and in applications needing access to information about the attributes of a type (for example, to convert them to a portable form for storage). The types and macros in this section are defined in the header `<stdint.h>`.

3.1 precof Operator

The `precof` operator returns an object of type `prec_t` (an integral type defined in `<stdint.h>`) that encodes the type attributes of an integral type or integral expression. The expression is not evaluated. Objects of type `prec_t` may be supplied as parameters to the `printf` and `scanf` family of Standard library functions, as well as the `EIR_*` macros. For any two integral types, *T1* and *T2*, *T1* and *T2* have the same integer specification (specified precision, actual precision, exactness, performance, signedness) if and only if `precof(`*T1*`) == precof(`*T2*`)`.

3.2 EIR_* Macros

There following macros exact information from an object of type `prec_t`. They are used in combination with the `precof` operator to determine the type attributes of a type or expression.

`EIR_BIT(`*prec_t t*`)`	Returns the number of bits of actual precision for *t*.
`EIR_SBIT(`*prec_t t*`)`	Returns the number of bits of specified precision for *t*.
`EIR_UNSIGNED(`*prec_t t*`)`	Returns 1 if *t* is a unsigned type, 0 otherwise.
`EIR_EXACT(`*prec_t t*`)`	Returns 1 if *t* is an exact type, 0 otherwise.
`EIR_TPERF(`*prec_t t*`)`	Returns >0 if *t* is optimized for time, 0 if *t* otherwise.
`EIR_SPERF(`*prec_t t*`)`	Returns >0 if *t* is optimized for space, 0 if *t* otherwise.

All of the macros return values of type `int`. The returned values are integral constants, and therefore may be used in integral constant expressions, such as specifying the precision of a type. The following example demonstrates multiplication without loss of precision.

```
typedef ... fact1_t;
typedef ... fact2_t;

#define fact1_len EIR_BIT(precof(fact1_t))
#define fact2_len EIR_BIT(precof(fact2_t))
#define prod_len (fact1_len+fact2_len)

signed int atleast:prod_len
my_prod(fact1_t f1, fact2_t f2)
{
        signed int atleast:prod_len p;
        p = f1;
        p *= f2;
        return p;
}
```

4. LIBRARY SERVICES

The introduction of additional integral types requires changes to the library services for formatted input and output. These changes will not affect existing conforming and strictly conforming programs.

4.1 printf

This proposal adds to the existing `printf` format specification syntax an optional `?` character specifying that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an integer whose type is specified by an argument of type `prec_t` immediately preceding the integral value in the argument list. The argument of type `prec_t` must be the result of applying the `precof` operator to a type or expression of the same type as integral value which immediately follows in the argument list, otherwise the result undefined.

4.2 scanf

This proposal adds to the existing `scanf` format specification syntax an optional `?` character indicating the size of the receiving object for conversion specifiers `d`, `i`, `o`, `u`, `x`, or `n` is specified by an argument of type `prec_t` immediately preceding the pointer to the receiving object in the argument list. The argument of type `prec_t` must be the result of applying the `precof` operator to a type or expression of the same type as the receiving object otherwise the result undefined.

4.3 strtoint

Synopsis

```
#include<stdint.h>
void *strtoint(const char *nptr, char **endptr,
        int base, void *buf, prec_t type);
```

Description

The `strtoint` converts the initial portion of the string pointed to by `nptr` to the representation specified by `type`. The conversion is performed as described for `strtol`. The result is stored in `buf`. The function returns a pointer to the converted value. If no conversion could be performed, a null pointer is returned.

5. CHANGES TO STANDARD C

The following wording changes are made to Standard C.

5.1 Types

Editing Notes

In subclause 6.1.2.5, add to the end of the fourth paragraph the following.

Text

(The range of extended integer types is described in 6.5.2.)

5.2 Integer constants

Editing Notes

In subclause 6.1.3.2, change the grammar for *integer-suffix* to;

integer-suffix:
    unsigned-suffix long-suffix-opt
    long-suffix unsigned-suffix-opt
    precision-suffix unsigned-suffix-opt

precision-suffix:
    `P` *digit-sequence*

Add to the end of the second paragraph in Semantics: ‘‘The optional precision suffix specifies the type of the constant is `unsigned exact:`*nn* if suffixed by the letters `u` or `U` and the type `signed exact:`*nn* if no suffix is appended.

5.3 Signed and unsigned integers

Editing Notes

In subclause 6.2.1.2, it the second and third paragraphs, replace ‘‘greater size’’ with ‘‘greater actual precision’’, and replace ‘‘smaller size’’ with ‘‘smaller actual precision’’.

5.4 Usual arithmetic conversions

Editing Notes

Add this in subclause 6.2.1.5. This text replaces the text starting with ‘‘If either operand has type `unsigned long int` ...’’ and ending with ‘‘Otherwise, both operands have type `int`.’’.

Text

— The integer specification is determined for both operands. The following are determined.

- Actual precision. The number of bits that participate in the value of the operand.
- Specified precision. The number of bits that were specified in the type definition of the operand.
- Exactness of precision. Whether the specified precision requires exactly *N* bits of precision, or a minimum (at least) *N* bits of precision.
- Performances attributes. Whether the type definition of the operand requires optimization for speed, optimization for time, or no optimization.
- Signedness. The operand is unsigned or signed.

— Operands whose actual precision is smaller than the actual precision of `int` are promoted to `int`.

— The actual precision of the result is at least the maximum of the actual precisions of the operands.

— If both operands are **exact**, the result is **exact**. Otherwise the result is **at-least**.

— The specified precision of the result must be at least the maximum of the actual precision of the operands.

— If both operands are signed, the result is signed. If bother operands are unsigned, the result is unsigned. Otherwise, if the operands differ in actual precision, the result is the the signedness of the operand with the larger actual precision. Otherwise, the result is unsigned.

— The result has no performance attributes.

— Both operands are promoted to the type of the result, then the operation is performed.

Editing Notes

The last sentence of footnote 29 should be replaced with the following text.

Text

Thus, the range of portable floating values is (-1,`EIR_MAX(precof(`*type*`))`+1).

5.5 Unary arithmetic operators

Editing Notes

In subclause 6.3.3.3, the third paragraph in Semantics, replace ‘‘The expression `~E` is equivalent ... `<limits.h>`.)’’ with the following.

Text

The expression `~E` is equivalent to `(EIR_MAX(precof(E))-E)`. (The macro `EIR_MAX` is defined in `<stdint.h>`.)

5.6 The precof operator

Editing Notes

Add this after subclause 6.3.3.4:

Constraints

The `precof` operator shall not be applied to an expression that has function type or non-integral type, or to the parenthesized name of such a type.

Semantics

The `precof` operator yields the precision information of its operand, which may be an expression or the parenthesized name of a type. The precision is determined from the type of the operand, which is not itself evaluated. The result is an integer constant.

Two types `T1` and `T2` shall have the same integer specification (specified precision, actual precision, exactness of precision, performance attributes, and signedness) if and only if `precof(T1) == precof(T2)`.

5.7 Bitwise shift operators

Editing Notes

In subclause 6.3.7, the second paragraph in Semantics, replace ‘‘reduced modulo `ULONG_MAX+1` ... `<limits.h>`.)’’ with the following.

Text

[...] reduced modulo `EIR_MAX(precof(E1))+1`. (The macro `EIR_MAX` is defined in `<stdint.h>`.)

5.8 Type specifiers

Editing Notes

In subclause 6.5.2, add the following to the Syntax.

Text

`fast`
`small`
`exact:`*constant-expression*
`atleast:`*constant-expression*

Editing Notes

Add the following to the dashed list in Constraints.

Text

— `signed int atleast:`*N*

— `signed fast int atleast:`*N*

— `signed small int atleast:`*N*

— `signed int exact:`*N*

— `signed fast int exact:`*N*

— `signed small int exact:`*N*

— `unsigned int atleast:`*N*

— `unsigned fast int atleast:`*N*

— `unsigned small int atleast:`*N*

— `unsigned int exact:`*N*

— `unsigned fast int exact:`*N*

— `unsigned small int exact:`*N*

Editing Notes

Add the following after the dashed list in Constraints.

Text

The actual precision of two integral types, differing otherwise only in their signedness, must be the same.

The actual precision of two integral types, differing otherwise only in their performance attribute, must be such that the actual precision of a type optimized for space is no greater than the actual precision of an unoptimized type, and the actual precision of an unoptimized type is no greater than the actual precision of a type optimized for time. Footnote: In other words, the ordering from smallest to largest actual precision is: `small` <= unoptimized <= `fast`.

The actual precision of two unoptimized integral types, differing otherwise only in their specified precision, must be such that the type with greater specified precision has actual precision no less than than the actual precision of the other type. Footnote: In other words, if *specified(N) >= specified(M)*, then *actual(N) >= actual(M)*.

Editing Notes

Add the following text prior to ‘‘Forward references’’.

Text

The *constant-expression* in extended integer types specifies the precision. Extended integer types that have different specified precision designate different types. `fast` indicates an implementation-defined optimization for time. `small` indicates an implementation-defined optimization for space. `atleast` indicates that the specified precision is the minimum number of bits that participate in its value. `exact` indicates that the specified precision is exactly the number of bits that participate in its value.

5.9 Extended Integer Range <stdint.h>

Editing Notes

Add this after subclause 7.1.6:

Text

The following types and macros are defined in the standard header `<stdint.h>`.

The type

`prec_t`

is the unsigned integral type of the result of the `precof` operator.

The macros are

`EIR_LG2FLOOR(`*v*`)`

which returns the largest integer less than or equal to the base 2 logarithm of the positive value *v*; and

`EIR_LG2CEIL(`*v*`)`

which returns the smallest integer greater than or equal to the base 2 logarithm of the positive value *v*; and

`EIR_MIN(`*prec_t t*`)`

which returns the minimum value of an object with precision specification *t*; and

`EIR_MAX(`*prec_t t*`)`

which returns the maximum value of an object with precision specification *t*; and

`EIR_BIT(`*prec_t t*`)`

which returns the number of bits of actual precision for *t*; and

`EIR_SBIT(`*prec_t t*`)`

which returns the number of bits of specified precision for *t*; and

`EIR_UNSIGNED(`*prec_t t*`)`

which returns 1 if *t* is a unsigned type, 0 otherwise; and

`EIR_EXACT(`*prec_t t*`)`

which returns 1 if *t* is an exact type, 0 otherwise; and

`EIR_TPERF(`*prec_t t*`)`

which returns a positive integer if *t* is optimized for time, 0 if *t* otherwise; and

`EIR_SPERF(`*prec_t t*`)`

which returns a positive integer if *t* is optimized for space, 0 if *t* otherwise.

5.10 The fprintf function

Editing Notes

Add this in subclause 7.9.6.1, prior to the dash list item ‘‘A character that specifies the type of conversion to be applied.’’.

Text

An optional `?` character specifying that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an integer whose type is specified by an argument of type `prec_t` immediately preceding the integral value in the argument list. The argument of type `prec_t` must be the result of applying the `precof` operator to a type or expression of the same type as integral value which immediately follows in the argument list, otherwise the behavior undefined.

5.11 The fscanf function

Editing Notes

Add this in subclause 7.9.6.2, prior to the dash list item ‘‘A character that specifies the type of conversion to be applied ...’’.

Text

An optional `?` character indicating the size of the receiving object for conversion specifiers `d`, `i`, `o`, `u`, `x`, or `n` is specified by an argument of type `prec_t` immediately preceding the pointer to the receiving object in the argument list. The argument of type `prec_t` must be the result of applying the `precof` operator to a type or expression of the same type as the receiving object otherwise the behavior undefined.

5.12 The strtoint function

Editing Notes

Add this after subclause 7.10.1.6.

Synopsis

```
#include <stdint.h>
void *strtoint(const char *nptr, char **endptr,
        int base, void *buf, prec_t type);
```

Description

The `strtoint` converts the initial portion of the string pointed to by `nptr` to the integral type described by `type`. The conversion is performed as described for `strtol`.

Returns

The result is stored in the object `buf` points to. The function returns a pointer to the converted value. If no conversion could be performed, a null pointer is returned. If the correct value is outside that range of representable values, no conversion is performed, a null pointer is returned, and the value of the macro `ERANGE` is stored in `errno`.

6. ISSUES AND RESOLUTION

6.1 Resolved Issues

The following issues have been resolved.

- The promotion rules are too complex with the propagation of EIR attributes. Resolution: Promotion is only based upon actual precision. The performance attributes doesn't propagate into expressions.
- The `strtoint` shouldn't call `malloc` is the `buf` pointer is null. Resolution: `strtoint` no longer calls `malloc`.
- How are constants specified? Resolution: The `P`*nn* precision suffix may be used in integral constants.
- The signedness of the promotion rules isn't value preserving. Resolution: The promotion rules have been simplified. In summary, the signedness of the result is the signedness of the operand with the larger actual precision.
- Can the **exact** type be removed? Resolution: No. Implementations would need to resort to bit masking for arbitrary precision values. Since the **exact** type is just a container around a structure bit field (the `&` and `sizeof` operators can be applied to the container), the compiler already supports these features.
- The paper talks about strictly conforming and conforming code, but this doesn't imply using specified precision versus actual precision. Resolution: The terms minimalist style (makes no assumptions about the implementation) and adaptive style (takes advantage of implementation limits) have been added.

6.2 Open Issues

The following issues are still outstanding.

- What is the minimum precision an implementation must support?
- How can this proposal be merged with the Kwan `inttypes.h` proposal.
- How can this be taught easily?
- The syntax is ugly.
- Are there implementations of this available? Partial Resolution: IEEE 1596.5 support integer types of 8, 16, 32, 64, and 128 bits. However, a sample compiler would be a better demonstration.