Farance Inc.

subject: **Extensions to <inttypes.h> Revision 1**
document: **WG14/N526 X3J11/95-127**
file: **`extended-integers/eir.*`**

date: **1995-12-22**

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

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

ABSTRACT

This proposal addresses the extended integer problem using the problem analysis of SBEIR proposal, but frames the solution around the <inttypes.h> design, i.e., using preprocessor features. This technique allows the programmer to specify extended integer types via requirements (minimum number of bits, space-time optimization), but uses the *preprocessor* to achieve this, thus, greatly reducing the complexity of the solution. This proposal allows the introduction of `long long` type as a C type because the primary objection to it (increases portability cost) can now be eliminated (hiding it with a `precis` macro). In summary, this solution provides several features: it uses existing C types, it provides a mechanism for specifying types based on requirements, it allows the use of `long long`, it provides promotion rules, it provides support for `printf` and `scanf`.

CONTENTS

6/7

1. OVERVIEW

This proposal is a fresh approach to addressing the extended integer problem. Although this proposal acknowledges the problems addressed by the SBEIR (Specification-Based Extended Integer Range) proposal, the solution is a completely new design. In short, this proposal is a compromise of several proposals: the <inttypes.h> proposal, the SBEIR proposal, and `long long` proposals. The following are a summary of the features:

1. Extended integer types can be described by a set of requirements (minimum precision, performance attributes) that *greatly* reduce porting cost across many platforms. The programmer is still free to use `char`, `short`, `int`, and `long`.
2. The simplicity of the <inttypes.h> solution is maintained by keeping most of the work in the preprocessor. **No new types are required.** The preprocessor maps the features to *existing* C types.
3. Additional types, such as `long long`, can be added without porting problems because there is a `precis()` feature. Thus, the `long long` extension that many vendors have implemented can remain as is.
4. With the generic promotion rules, de facto types, such as `long long`, or vendor-specific types, e.g., `int48`, can be included in implementations because programmers will know how they interact.
5. C++ overloading needs are addressed by **not** creating new types: the `precis()` does a preprocessor map to existing C types.

2. FEATURES

2.1 precis() macro

In previous papers, it was demonstrated that ''loss of information causes portability problems'' for extended integer range types. The primary problem has been mapping the programmer's requirements to the capabilities of the target system, e.g., if the ''fastest type of 32 bits'' is the requirement, what type does that map into in C? Previous proposals dealt with, literally, creating new types in C.

The `precis()` macro is used to specify the precision for the type. The precision specified is a *minimum* (i.e., ''at least'') requirement for precision. For example:

```
main()
{
        /* at least 32 bits of precision */
        precis(32) X,Y,Z;

        X = Y+Z;
}
```

The performance attributes can be included by OR-ing features in, as necessary:

```
struct
{
        /*
         * Space optimization: smallest storage
         * when using structures to be saved in
         * external storage
         */
        precis(32|INT_SMALL_C) A;
} B;
```

```
my_api
(
        /*
         * The API (the interface) doesn't require
         * space-time optimization.
         */
        precis(32) E;
}
{
        /*
         * Time optimization: The fastest version
         * of the type is used for ''inner'' loops
         * to speed calculations.
         */
        precis(32|INT_FAST_C) F,G;

        G = 0;
        F = E;

        while ( -F >= 0 )
        {
                G += F;
        }

        /*
         * The result is stored in the smallest
         * type that matches the requirements.
         */
        B.A = G;
}
```

Since `precis()` is a macro and it expands to one of the *existing* C types, there is no new syntax or typing required. In other words, the main problem (''the loss of information causes portability problems'') is solved without major changes to the language. The qualifier `signed` or `unsigned` can be added as:

```
typedef unsigned precis(32|INT_FAST_C) ufast32;
```

An important question is: How to implement `precis()` as a macro? The following might be some implementation of the `precis()` macro.

```
/*
 * Contents of "<stdint.h>" for
 * a sample 32/64-bit machine.  In this
 * implementation, the "SMALL" attribute
 * has no effect on the choice of types.
 */

#define precof(P) ((sizeof(P))*CHAR_BIT)

#define INT_FAST_C 0x1000
#define INT_SMALL_C 0x2000
#define INT_PREC 0xFFF

/*
 * On this machine, any ``fast'' type greater
 * than 16 bits gets mapped to "long".  Otherwise,
 * the type is mapped to "short".
 *
 * For unoptimized and ``small'' types, >32 is mapped
 * to "long", >16 is mapped to "int", >8 is mapped to
 * "short", and <= 8 is mapped to "char".
 */
#define precis(P) \
#if ( P & INT_FAST_C ) /* choose fast type */
        #if ( ((P) & INT_PREC) > 16 )
                long
        #else
                short
        #endif
#else /* choose unoptimized or small type */
        #if ( ((P) & INT_PREC) > 32 )
                long
        #elif ( ((P) & INT_PREC) > 16 )
                int
        #elif ( ((P) & INT_PREC) > 8 )
                short
        #else
                char
        #endif
#endif
```

Note that the `#define` has a `#if` included in it. Currently, this is not permitted in the preprocessor. Either C9X would need to include this extension, or `precis` would have to be recognized specially in the preprocessor.

If the committee choses to extend the preprocessor, then we're not tied to a specific syntax of a preprocessor extension, only the capability to include `#if` *inside* a `#define`. For example, it might be acceptable to require the body of the `#define` above to have backslashes at the end of each line.

2.2 Performance Attributes

The following performance attributes are supported:

— `INT_FAST` — the fastest integer type (optional — this might not be a C type).

— `INT_FAST_C` — the fastest integer type that is a C type (required).

— `INT_SMALL_C` — the smallest type that is a C type (required).

— `INT_SMALL_BYTE` — the smallest type that fits in a byte boundary (optional — this might not be a C type).

— `INT_SMALL_BIT` — the smallest type that fits in a bit field, contained in an addressable unit (optional — this might not be a C type, nor possible on some systems).

2.3 <inttypes.h> compatibility

Each of the `precis` types could be derived from the <inttypes.h> types or vice versa. The following is an example if implementing the `precis` types via existing <inttypes.h> types.

```
/*
 * Contents of "<stdint.h>" for
 * a machine uses "<inttypes.h>".  In this
 * implementation, the "SMALL" attribute
 * has no effect on the choice of types.
 */

#define precof(P) ((sizeof(P))*CHAR_BIT)

#define INT_FAST_C 0x1000
#define INT_SMALL_C 0x2000
#define INT_PREC 0xFFF

/*
 * On this machine, any ``fast'' type uses
 * the type "intfast_t".  All other types map
 * to appropriate "<inttypes.h>" types.
 */
#define precis(P) \
#if ( P & INT_FAST_C ) /* choose fast type */
        intfast_t
#elif ( P & INT_SMALL_C ) /* choose small type */
        #if ( ((P) & INT_PREC) > 32 )
                int_least64_t
        #elif ( ((P) & INT_PREC) > 16 )
                int_least32_t
        #elif ( ((P) & INT_PREC) > 8 )
                int_least16_t
        #else
                int_least8_t
        #endif
#else /* choose unoptimized type */
        #if ( ((P) & INT_PREC) > 32 )
                int64_t
        #elif ( ((P) & INT_PREC) > 16 )
                int32_t
        #elif ( ((P) & INT_PREC) > 8 )
                int16_t
        #else
                int8_t
        #endif
#endif
```

2.4 precof() macro

The `precof()` operator extracts precision information, necessary for `printf` or `scanf`.

```
typedef precis(32|INT_FAST_C) fast32;

func()
{
        fast32 X;

        printf("X has value: %?d\n",precof(X),X);
}
```

On many systems, `precof` can be implemented as:

```
#define precof(X) ((sizeof X)*CHAR_BIT)
```

2.5 Integer Constants

The programmer can use constants with programmer-specified precision. The notation is similar to the exponent notation for floating constants: the `I`*nn* suffix specifies an integer constant of *nn* bits. For example, `123I16` specifies a signed, 16-bit constant; `456I32U` specifies an unsigned, 32-bit constant. Some C compilers already provide constants like these.

2.6 long long type

The `long long` and `unsigned long long` types are added to provide a *minimum* of 64 bits of precision.

2.7 Preprocessor Constants

The preprocessor must support integral constants up to a minimum of the precision of `long long` and `unsigned long long`. This means that the preprocessor must support at least 64-bit arithmetic.

2.8 printf and scanf support

This proposal adds to the existing `printf` and `scanf` format specification syntax an optional `?` character specifying that a following `d`, `i`, `o`, `u`, `x`, `X`, or `u` conversion specifier applies to an integer whose type is specified by `precof()` argument immediately preceding the integral value in the argument list.

Additionally, the `L` qualifier is used to support `long long`. For example:

625

```
/*
 * Contents of "<stdint.h>" for
 * a machine uses "<inttypes.h>".  In this
 * implementation, the "SMALL" attribute
 * has no effect on the choice of types.
 */

#define precof(P) ((sizeof(P))*CHAR_BIT)

#define INT_FAST_C 0x1000
#define INT_SMALL_C 0x2000
#define INT_PREC 0xFFF

/*
 * On this machine, any ``fast'' type uses
 * the type "intfast_t".  All other types map
 * to appropriate "<inttypes.h>" types.
 */
#define precis(P) \
#if ( P & INT_FAST_C ) /* choose fast type */
        intfast_t
#elif ( P & INT_SMALL_C ) /* choose small type */
        #if ( ((P) & INT_PREC) > 32 )
                int_least64_t
        #elif ( ((P) & INT_PREC) > 16 )
                int_least32_t
        #elif ( ((P) & INT_PREC) > 8 )
                int_least16_t
        #else
                int_least8_t
        #endif
#else /* choose unoptimized type */
        #if ( ((P) & INT_PREC) > 32 )
                int64_t
        #elif ( ((P) & INT_PREC) > 16 )
                int32_t
        #elif ( ((P) & INT_PREC) > 8 )
                int16_t
        #else
                int8_t
        #endif
#endif
```

- Provide a sample implementation of the preprocessor extension on the WG14 FTP site.