Preprocessor improvements for better configuration possibilities and increased source code portability with performance critical systems.

Proposal for a work item

1. Scope

As the C language has matured over the years and the C compilers have become better to generate efficient code it has become common practice to write performance critical code in C. With the increased use of embedded processor in electronic products, and the tendency towards making multiple version of the same product, there is a increasingly demand for making performance critical source code adaptable to multiple execution environments.

Configuration based on memory instantiated parameters causes server penalties in the form of increased execution time and memory footprint. Such a configuration method is usually not an acceptable solution in performance critical systems. Instead the right driver functionality must be picked at compile time based on defined configuration parameters.

This paper propose a new work item for investigating how better support for compile time configuration can be implemented in the language.

2. Rationale

The C standard currently lacks good features for selecting the right code based on configuration parameter definitions without any runtime penalty.

Especially the standard lacks features for efficient parameter based selection at compile time between different function implementations within the same compilation unit.

To illustrate this and in order to find a possible solution let us first consider this example:

Example 1

```
#define uint16_t int
char array1[4] = {0x12,0x34,0x56,0x78};

// Fetch method 1
inline uint16_t use_method_1( int index )
{
   return array1[index] * 256 + array1[index+1];
}
```

```
// Fetch method 2
uint16_t use_method_2( int index, int size)
{
   return (array1[(index * size)+1] * 256 + array1[index * size]);
}


// Select method using a function parameter
uint16_t getval(char method, int idx)
{
  if (method == 0)
     return use_method_1( idx );
  else
     return use_method_2( idx, 2 );
}

#define  METHOD1 0
#define  METHOD2 1
uint16_t val[4];

void my_func(void)
{
   val[0] = getval(METHOD1, 0);  // 0x1234
   val[1] = getval(METHOD1, 1);  // 0x3456
   val[2] = getval(METHOD2, 0);  // 0x3412
   val[3] = getval(METHOD2, 1);  // 0x7856
}
```

Selection between different code fragments (different functions) is based on some configuration parameters evaluated at runtime. The problem with respect to performance is the code size and time overhead caused by each of the *getval* function calls.

This is an unacceptable solution in performance critical systems.

Execution performance in the example above could be optimized with respect to execution speed by making *getval* inline or by converting it to a function macro like:

Example 2

```
// Version selected by a macro parameter
#define getval(method, idx)         \
(                                   \
  (method == 0) ?                   \
     use_method_1( idx )            \
  :                                 \
     use_method_2( idx, 2 )         \
)
```

Inline functions and macros are inadequate
Approaches like example 2 will however often create the problem that “dead code” is generated in every instantiation of the macro for the configuration alternatives not selected by the configuration parameter.

This is an unacceptable solution in many embedded systems where small memory foot prints a requirement. (The programmer must rely on that the compiler has an aggressive optimizer which is able to remove dead code)

3. A possible solution

For the discussion lets assume that this code is legal:

```
Example 3:
   // Version selected by a function parameter
   #define getval(method, idx) \
   (                           \
     #if (method == 0)         \
        use_method_1( idx)     \
     #else                     \
        use_method_2( idx,2 )  \
     #endif                    \
   )
```

The advantages with solutions along this line are that we get the best of several worlds:

- Optimal execution speed.
- Prevention of dead code generation at the source level (instead of relying on compiler optimization)
- With #if, #else, #endif having macro scope different code alternatives can be selected by configuration parameters in each macro instantiation.

Another advantage by this approach is that external modules with “dead code” does not have to exist just to satisfy the linker:

```
Example 4

   // Fetch method 1
   inline uint16_t use_method_1( int index )
   {
      return array1[index] * 256 + array1[index+1];
   }

   // Fetch method 2
   extern uint16_t use_method_2( int index, int size)

   // Version selected by a function parameter
   #define getval(method, idx) \
   {                           \
     #if (method == 0)         \
        use_method_1( idx);    \
     #else                     \
        use_method_2( idx,2 ); \
     #endif                    \
   }

   #define  METHOD1 0
   #define  METHOD2 1
   uint16_t val[4];

   void my_func(void)
   {
      val[0] = getval(METHOD1, 0);  // 0x1234
      val[1] = getval(METHOD1, 1);  // 0x3456
   }
```

Only METHOD1 is used. If *getval* had been implemented as in example 2 then *use_method_2()* had to be defined even though it is never used by the actual configuration.

The macro approach also makes compile time configuration much more elegant as individual configuration parameters for each “instantiation” of the same macro can be collected in a single configuration header file.

```
Example 5

// Configuration header file
#define  DEVICE1_METHOD     0
#define  DEVICE1_CONSTANT   1

#define  DEVICE2_METHOD     1
#define  DEVICE2_CONSTANT   0
#define  DEVICE2_CONSTANT1  2

#define  DEVICE3_METHOD     0
#define  DEVICE3_CONSTANT   2

// header file
#define getval(device)                  \
(                                       \
  #if (device##_METHOD == 0)            \
     use_method_1( device##_CONSTANT )  \
  #else                                 \
     use_method_2( device##_CONSTANT,   \
                   device##_CONSTANT1)  \
  #endif                                \
)

// Driver module
void my_func(void)
   {
   val[0] = getval(DEVICE1);    // use_method_1, inlined
   val[1] = getval(DEVICE2);    // use_method_2, function call
   val[2] = getval(DEVICE3);    // use_method_1, inlined
   }
```

Furthermore the configuration parameters and configuration dependent code can be completely isolated from the driver code itself. This enable new methods to be used without the need to make modification to the driver source code itself. Maintenance therefore become much simpler.

Jan Kristoffersen
Member of Danish Standard