Document Number: WG14 N478/X3J11 95-079

C9X Revision Proposal
=====================

Title: New Machine Characteristics macros
Author: David R. Tribble
Author Affiliation: Self
Postal Address: 6004 Cave River Dr.
Plano, TX 75093-6951
USA
E-mail Address: drt@wcwcen.wf.com
Telephone Number: +1 214 964 1720 CST
Fax Number:
Sponsor:
Date: 1995-05-08
Proposal Category:
__ Editorial change/non-normative contribution
__ Correction
X_ New feature
__ Addition to obsolescent feature list
__ Addition to Future Directions
__ Other (please specify) ________________________
Area of Standard Affected:
__ Environment
__ Language
__ Preprocessor
__ Library
X_ Macro/typedef/tag name
__ Function
X_ Header
__ Other (please specify) ________________________
Prior Art: None.
Target Audience: Programmers and configuration managers who port C programs to several platforms.
Related Documents (if any):
Proposal Attached: X_ Yes __ No, but what's your interest?
Abstract: The task of writing portable code for several platforms is easier in a language like C. Information is available (from header files like <limits.h>) that allow programs to be written for different CPUs. However, more information would be useful. Such info would specify machine (CPU) characteristics, operating system info, and compiler info.

===================== Cover sheet ends here =============

PROPOSAL

New constants (preprocessor macros) will be added to the standard header files which specify characteristics of the implementation, including the compiler, the operating system, and the underlying CPU hardware.

RATIONALE

C provides a good framework for writing very portable code (despite its reputation for being a ''low-level'' language). Information contained in the "<limits.h>" and "<float.h>" headers allows programs to be written with conditionally compiled code to address platform differences, and thus be very portable.

However, each implementation of C and each CPU architecture has characteristics that make it sufficiently different from other implementations, making things difficult for programmers to produce truly portable code. Some specification information just isn't available to programs; this information typically must be supplied at compile time, either through command line options to the compiler or by editing certain #include files. It would be nice if information about these characteristics was available in a standard way, so that code could be aware of and make allowances for the differences.

The classic example of an implementation-specific characteristic is byte ordering within words (i.e., big-endian versus little-endian). This information is deducible programmatically, but it would be far more useful to have this available to the program in the form of a constant or data structure in a standard header file.

IMPLEMENTATION

The addition of new constants (macros) in the standard header files which specify the implementation-specific characteristics of the compiler, operating system, and CPU architecture, would give the programmer's code enough knowledge so that it could make whatever allowances are necessary to make it port and run correctly on a wide variety of machines. This would also allow code generating programs that produce C code to tailor the generation to the target machine.

The constants would be #defined macro names, and would begin with an underscore prefix ('_') (since names with leading underscores are reserved for each implementation [7.1.3]). A required set of names would be delineated for each header file, while allowing implementors to add extra constants to the headers.

A program can then #include the appropriate header(s), and use the constants. The program can also test for certain features values using '#if' directives. A program can test for the presence of a feature in addition to using the specific feature's actual value.

Unknown or inapplicable characteristics would be coded as zero (0) or empty string ("") values, or not be #defined at all.

NEW CONSTANTS

The header files affected are:

    <float.h>
    <limits.h>

Additions to each header are described below.

[If adding the new constants to existing standard header files is considered ill-advised, the alternative is to define new header files which contain the new functions, such as '<cpu.h>', '<os.h>', and '<compiler.h>'.]

CPU-SPECIFIC CONSTANTS

The <limits.h> header would contain constants describing characteristics of the target CPU, such as word sizes, byte order, data type alignment restrictions, etc. It would also contain constants defining the name (type) of the CPU and the name of the CPU manufacturer (vendor).

The new constants (with example values) are:

```
/* CPU type */

#define _CPU_NAME        "iAPX/386" /* Name/type        */
#define _CPU_FAMILY      "iAPX"     /* Family           */
#define _CPU_MOD         "DX2"      /* Modification     */
#define _CPU_VERS        "1.0.00"   /* Version          */
#define _CPU_VENDOR      "Intel"    /* Vendor name      */

/* Bit/byte/word order */

#define _ORD_BIG         0         /* Big-endian                 */
#define _ORD_LITTLE      1         /* Little-endian              */

#define _ORD_BITF_HL     0         /* Bitfield fill order        */
#define _ORD_BYTE_HL     0         /* Byte order within shorts   */
#define _ORD_WORD_HL     0         /* Word order within longs    */

/* Data type bit sizes */

#define _BITS_BITF_MIN  8          /* Min bits in bitfield */
#define _BITS_BITF_MAX 32          /* Max bits in bitfield */

#define _BITS_CHAR       8         /* Char         */
#define _BITS_SHRT       16        /* Short        */
#define _BITS_INT        16        /* Int          */
#define _BITS_LONG       32        /* Long         */

#define _BITS_FLT        32        /* Float        */
#define _BITS_DBL        64        /* Double       */
#define _BITS_LDBL       80        /* Long double  */

#define _BITS_PTR        16        /* Pointer      */
#define _BITS_ADDR       16        /* Address range*/

/* Data type alignments */

#define _ALIGN_CHAR      1         /* Char         */
#define _ALIGN_SHRT      2         /* Short        */
#define _ALIGN_INT       2         /* Int          */
#define _ALIGN_LONG      4         /* Long         */
#define _ALIGN_FLT       4         /* Float        */
#define _ALIGN_DBL       8         /* Double       */
#define _ALIGN_LDBL      4         /* Long double  */
#define _ALIGN_PTR       2         /* Pointer      */

/* Data type signed-ness */

#define _UBITF           0         /* Plain bitfield is unsigned  */
#define _UCHAR           0         /* Plain char is unsigned      */
```

Optional constants would be added by implementors for characteristics peculiar to the hardware.

For example, MS-DOS programs running on 16-bit Intel 8086 CPUs, with its segmented addressing, might include constants such as:

```
#define _BITS_NPTR      16        /* Near pointer          */
#define _BITS_FPTR      32        /* Far pointer           */
#define _BITS_NADDR     16        /* Near address range    */
#define _BITS_FADDR     16        /* Far address range     */

#define _ALIGN_NPTR     2         /* Near pointer */
#define _ALIGN_FPTR     2         /* Far pointer  */
```

As another example, compilers that supported a 64-bit 'long long int' data type would add:

```
#define _ORD_LONG_HL    0         /* Word order within long long  */

#define _BITS_LLONG     64        /* Long long    */

#define _ALIGN_LLONG    4         /* Long long    */
```

Another example: operating systems with character sets other than 7-bit ASCII, such as 8-bit EBCDIC:

```
#define _UCHAR          1         /* Plain char is unsigned */
```

FLOATING-POINT CONSTANTS

Additions to the existing header file <float.h> [5.2.4.2.2] include constants specifying floating-point representations for infinity, denormalized values, IEEE compliance, etc.

These constants would be defined as true (1) or false (0).

The new constants (with example values) are:

```
/* IEEE-compliant floating-point formats */

#define FLT_IEEE    1   /* Is IEEE compliant */
#define DBL_IEEE    1
#define LDBL_IEEE   1

/* Floating-point infinity values */

#define FLT_INF     1   /* Has infinity */
#define DBL_INF     1
#define LDBL_INF    1

#define FLT_NINF    1   /* Has negative infinity */
#define DBL_NINF    1
#define LDBL_NINF   1

/* Floating-point not-a-number values */

#define FLT_NAN     1   /* Has NaN */
#define DBL_NAN     1
#define LDBL_NAN    1

/* Floating-point denormals */
```

```
#define FLT_DENORM  1   /* Has denormals */
#define DBL_DENORM  1
#define LDBL_DENORM 1
```

OPERATING SYSTEM-SPECIFIC CONSTANTS

The <limits.h> header contains constants that specify characteristics about the target operating system, including the name, vendor, date and time it was built/released, etc.

The new constants (with example values) are:

```
/* Operating system info */

#define _OS_NAME        "Unix"          /* Name         */
#define _OS_VERS        "5.4.01"        /* Version      */
#define _OS_REL         5               /* Release      */
#define _OS_LEV         4               /* Level        */
#define _OS_UPD         1               /* Update       */
#define _OS_DATE        "DD Mon YYYY"   /* Release date */
#define _OS_TIME        "HH:MM:SS"      /* Release time */
#define _OS_VENDOR      "Company"       /* Vendor name  */

#define _OS_ASCII       1               /* Uses ASCII   */
```

Other character set constants might include:

```
#define _OS_EBCDIC      1               /* EBCDIC       */
#define _OS_ISO646      1               /* ISO 646      */
#define _OS_ISO10646    1               /* ISO 10646    */
#define _OS_JIS         1               /* JIS ASCII    */
#define _OS_EUC         1               /* EUC ASCII    */
#define _OS_UNICODE     1               /* Unicode      */
```

[This is tricky. How does an operating system indicate which character set it uses with a simple constant?]

COMPILER-SPECIFIC CONSTANTS

The <limits.h> header contains constants that specify information about the compiler, including the version number, vendor, date and time it was built/released, etc. It also contains preprocessor and compiler limitations, such as the maximum levels of nested #includes, etc.

Typical contents are:

```
/* Compiler info */

#define _COMPILER_NAME      "GNU CC"        /* Name                 */
#define _COMPILER_VERS      "1.2.03"        /* Version              */
#define _COMPILER_REL       1               /* Release              */
#define _COMPILER_LEV       2               /* Level                */
#define _COMPILER_UPD       3               /* Update               */
#define _COMPILER_DATE      "DD Mon CCYY"   /* Build date           */
#define _COMPILER_TIME      "HH:MM:SS"      /* Build time           */
#define _COMPILER_VENDOR    "Company"       /* Vendor name          */
#define _COMPILER_LANG      198910L         /* C std vers, YYYYMM   */
#define _COMPILER_HOSTED    1               /* Is hosted environment*/
#define _COMPILER_CROSS     0               /* Is cross-compiler    */
```

```
/* Preprocessor and source code limits [5.2.4.1] */

#define _SOURCE_IFS         8       /* Max nested #ifs              */
#define _SOURCE_MPARMS      31      /* Max func macro parms         */
#define _SOURCE_WIDTH       509     /* Max chars per line           */
#define _SOURCE_STRING      509     /* Max chars per string const   */
#define _SOURCE_INCLUDES    8       /* Max nested #includes         */
#define _SOURCE_STMT        15      /* Max stmt nesting levels      */
#define _SOURCE_TYPES       12      /* Max declarators per type     */
#define _SOURCE_DECLS       31      /* Max nested declarators       */
#define _SOURCE_EXPR        32      /* Max nested expr              */
#define _SOURCE_IDENT       31      /* Max significant chars in name*/
#define _SOURCE_EXTERNID    6       /* Max signif chars in extern   */
#define _SOURCE_EXTERNS     511     /* Max extern names per file    */
#define _SOURCE_BLOCKID     127     /* Max names in a block         */
#define _SOURCE_MACROS      1024    /* Max macros per file          */
#define _SOURCE_FPARMS      31      /* Max func parms               */
#define _SOURCE_DATA        32767   /* Max bytes in an object       */
#define _SOURCE_CASES       257     /* Max case labels              */
#define _SOURCE_MEMBS       127     /* Max struct members           */
#define _SOURCE_ENUMS       127     /* Max enum constants per tag   */
#define _SOURCE_STRUCTS     15      /* Max nested structs           */
```

The '_COMPILER_LANG' value specifies the C language standard supported by the compiler. It is encoded as 'YYYYMM' for year and month numbers, and is a long int value.

Implementations that have no practical upper limit on a given value would #define the corresponding macro to be zero.

CONSTRAINTS

While it is possible to #define these constants to something other than a simple numeric or string literal, such as a library function call, it is preferable to constrain the definitions to simple constants.

One reason for this is so that programs can use the constants for conditionally compiled code (as operands of #if expressions), which require values to be resolvable during the preprocessing phase.

For example:

```
/* Integer type with at least 16 bits */

#if _BITS_SHRT >= 16

typedef signed short    int16;
typedef unsigned short  uint16;

#else

typedef signed int      int16;
typedef unsigned int    uint16;

#endif

/* Integer type with at least 32 bits */
```

```
#if _BITS_INT >= 32

typedef signed int       int32;
typedef unsigned int     uint32;

#else

typedef signed long      int32;
typedef unsigned long    uint32;

#endif
```

Another example:

```
/* hton() -- Convert 16-bit short into network-order short */

extern int  hton(int i);

#if _ORD_BYTE_HL
#define hton(i) (i)
#endif

/* lton() -- Convert 32-bit int into network-order int */

extern int  lton(int i);

#if _ORD_WORD_HL
#define lton(i) (i)
#endif
```

Another reason for constraining the macros to simple constants is to allow for the declaration of variables that are defined in terms of, or initialized to, one or more of the constants (which won't work if the macros are function calls).

For example:

```
static const char   target_cputype[] =  "CPU:" _CPU_NAME;
static const char   target_cpuvers[] =  "VS: " _CPU_VERS;
static const char   target_opsys[] =    "OS: " _OS_NAME;
static const char   source_compile[] =  "CC: " _COMPILER_NAME;
```

And yet another reason for constraining the macros is for efficiency (i.e., don't incur the overhead of a function call if it's not absolutely necessary).

Note that the macros could be #defined to built-in reserved words or identifiers that are specially known to the compiler.

Some plausible examples might be:

```
#define _COMPILER_LANG   __VERSION__

#define _ORD_BYTE_HL     __BIG_ENDIAN
#define _ORD_WORD_HL     __BIG_ENDIAN
```

================ END OF PROPOSAL ============================