```
/* Copyright (c) 2002, Marek Michalkiewicz
   Copyright (c) 2004,2007 Joerg Wunsch

   Portions of documentation Copyright (c) 1990, 1991, 1993, 1994
   The Regents of the University of California.

   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE.

  $Id$
*/

#ifndef _STDLIB_H_
#define _STDLIB_H_ 1

#ifndef __ASSEMBLER__

#define __need_NULL
#define __need_size_t
#define __need_wchar_t
#include <stddef.h>

#ifndef __ptr_t
#define __ptr_t void *
#endif

#ifdef __cplusplus
extern "C" {
#endif

/** \file */

/** \defgroup avr_stdlib <stdlib.h>: General utilities
    \code #include <stdlib.h> \endcode

    This file declares some basic C macros and functions as
    defined by the ISO standard, plus some AVR-specific extensions.
*/

/*@{*/
/** Result type for function div(). */
```

```
typedef struct {
  int quot;                     /**< The Quotient. */
  int rem;                      /**< The Remainder. */
} div_t;

/** Result type for function ldiv(). */
typedef struct {
  long quot;                    /**< The Quotient. */
  long rem;                     /**< The Remainder. */
} ldiv_t;

/** Comparision function type for qsort(), just for convenience. */
typedef int (*__compar_fn_t)(const void *, const void *);

#ifndef __DOXYGEN__

#ifndef __ATTR_CONST__
# define __ATTR_CONST__ __attribute__((__const__))
#endif

#ifndef __ATTR_MALLOC__
# define __ATTR_MALLOC__ __attribute__((__malloc__))
#endif

#ifndef __ATTR_NORETURN__
# define __ATTR_NORETURN__ __attribute__((__noreturn__))
#endif

#ifndef __ATTR_PURE__
# define __ATTR_PURE__ __attribute__((__pure__))
#endif

#ifndef __ATTR_GNU_INLINE__
# ifdef __GNUC_STDC_INLINE__
#  define __ATTR_GNU_INLINE__   __attribute__((__gnu_inline__))
# else
#  define __ATTR_GNU_INLINE__
# endif
#endif

#endif

/** The abort() function causes abnormal program termination to occur.
    This realization disables interrupts and jumps to _exit() function
    with argument equal to 1. In the limited AVR environment, execution is
    effectively halted by entering an infinite loop. */
extern void abort(void) __ATTR_NORETURN__;

/** The abs() function computes the absolute value of the integer \c i.
   \note The abs() and labs() functions are builtins of gcc.
*/
extern int abs(int __i) __ATTR_CONST__;
#ifndef __DOXYGEN__
#define abs(__i) __builtin_abs(__i)
#endif

/** The labs() function computes the absolute value of the long integer
    \c i.
   \note The abs() and labs() functions are builtins of gcc.
*/
extern long labs(long __i) __ATTR_CONST__;
#ifndef __DOXYGEN__
#define labs(__i) __builtin_labs(__i)
#endif

/**
     The bsearch() function searches an array of \c nmemb objects, the
```

```
     initial member of which is pointed to by \c base, for a member
     that matches the object pointed to by \c key.  The size of each
     member of the array is specified by \c size.

     The contents of the array should be in ascending sorted order
     according to the comparison function referenced by \c compar.
     The \c compar routine is expected to have two arguments which
     point to the key object and to an array member, in that order,
     and should return an integer less than, equal to, or greater than
     zero if the key object is found, respectively, to be less than,
     to match, or be greater than the array member.

     The bsearch() function returns a pointer to a matching member of
     the array, or a null pointer if no match is found.  If two
     members compare as equal, which member is matched is unspecified.
*/
extern void *bsearch(const void *__key, const void *__base, size_t __nmemb,
        size_t __size, int (*__compar)(const void *, const void *));

/* __divmodhi4 and __divmodsi4 from libgcc.a */
/**
     The div() function computes the value \c num/denom and returns
     the quotient and remainder in a structure named \c div_t that
     contains two int members named \c quot and \c rem.
*/
extern div_t div(int __num, int __denom) __asm__("__divmodhi4") __ATTR_CONST__;
/**
     The ldiv() function computes the value \c num/denom and returns
     the quotient and remainder in a structure named \c ldiv_t that
     contains two long integer members named \c quot and \c rem.
*/
extern ldiv_t ldiv(long __num, long __denom) __asm__("__divmodsi4") __ATTR_CONST__;

/**
     The qsort() function is a modified partition-exchange sort, or
     quicksort.

     The qsort() function sorts an array of \c nmemb objects, the
     initial member of which is pointed to by \c base.  The size of
     each object is specified by \c size.  The contents of the array
     base are sorted in ascending order according to a comparison
     function pointed to by \c compar, which requires two arguments
     pointing to the objects being compared.

     The comparison function must return an integer less than, equal
     to, or greater than zero if the first argument is considered to
     be respectively less than, equal to, or greater than the second.
*/
extern void qsort(void *__base, size_t __nmemb, size_t __size,
      __compar_fn_t __compar);

/**
    The strtol() function converts the string in \c nptr to a long
    value.  The conversion is done according to the given base, which
    must be between 2 and 36 inclusive, or be the special value 0.

    The string may begin with an arbitrary amount of white space (as
    determined by isspace()) followed by a single optional \c '+' or \c '-'
    sign.  If \c base is zero or 16, the string may then include a
    \c "0x" prefix, and the number will be read in base 16; otherwise,
    a zero base is taken as 10 (decimal) unless the next character is
    \c '0', in which case it is taken as 8 (octal).

    The remainder of the string is converted to a long value in the
    obvious manner, stopping at the first character which is not a
    valid digit in the given base.  (In bases above 10, the letter \c 'A'
    in either upper or lower case represents 10, \c 'B' represents 11,
```

```
    and so forth, with \c 'Z' representing 35.)

    If \c endptr is not NULL, strtol() stores the address of the first
    invalid character in \c *endptr.  If there were no digits at all,
    however, strtol() stores the original value of \c nptr in \c
    *endptr.  (Thus, if \c *nptr is not \c '\\0' but \c **endptr is \c '\\0'
    on return, the entire string was valid.)

    The strtol() function returns the result of the conversion, unless
    the value would underflow or overflow.  If no conversion could be
    performed, 0 is returned.  If an overflow or underflow occurs, \c
    errno is set to \ref avr_errno "ERANGE" and the function return value
    is clamped to \c LONG_MIN or \c LONG_MAX, respectively.
*/
extern long strtol(const char *__nptr, char **__endptr, int __base);

/**
    The strtoul() function converts the string in \c nptr to an
    unsigned long value.  The conversion is done according to the
    given base, which must be between 2 and 36 inclusive, or be the
    special value 0.

    The string may begin with an arbitrary amount of white space (as
    determined by isspace()) followed by a single optional \c '+' or \c '-'
    sign.  If \c base is zero or 16, the string may then include a
    \c "0x" prefix, and the number will be read in base 16; otherwise,
    a zero base is taken as 10 (decimal) unless the next character is
    \c '0', in which case it is taken as 8 (octal).

    The remainder of the string is converted to an unsigned long value
    in the obvious manner, stopping at the first character which is
    not a valid digit in the given base.  (In bases above 10, the
    letter \c 'A' in either upper or lower case represents 10, \c 'B'
    represents 11, and so forth, with \c 'Z' representing 35.)

    If \c endptr is not NULL, strtoul() stores the address of the first
    invalid character in \c *endptr.  If there were no digits at all,
    however, strtoul() stores the original value of \c nptr in \c
    *endptr.  (Thus, if \c *nptr is not \c '\\0' but \c **endptr is \c '\\0'
    on return, the entire string was valid.)

    The strtoul() function return either the result of the conversion
    or, if there was a leading minus sign, the negation of the result
    of the conversion, unless the original (non-negated) value would
    overflow; in the latter case, strtoul() returns ULONG_MAX, and \c
    errno is set to \ref avr_errno "ERANGE".  If no conversion could
    be performed, 0 is returned.
*/
extern unsigned long strtoul(const char *__nptr, char **__endptr, int __base);

/**
    The atol() function converts the initial portion of the string
    pointed to by \p s to long integer representation. In contrast to

        \code strtol(s, (char **)NULL, 10); \endcode

    this function does not detect overflow (\c errno is not changed and
    the result value is not predictable), uses smaller memory (flash and
    stack) and works more quickly.
*/
extern long atol(const char *__s) __ATTR_PURE__;

/**
    The atoi() function converts the initial portion of the string
    pointed to by \p s to integer representation. In contrast to

        \code (int)strtol(s, (char **)NULL, 10); \endcode
```

```
    this function does not detect overflow (\c errno is not changed and
    the result value is not predictable), uses smaller memory (flash and
    stack) and works more quickly.
*/
extern int atoi(const char *__s) __ATTR_PURE__;

/**
   The exit() function terminates the application.  Since there is no
   environment to return to, \c status is ignored, and code execution
   will eventually reach an infinite loop, thereby effectively halting
   all code processing.  Before entering the infinite loop, interrupts
   are globally disabled.

   In a C++ context, global destructors will be called before halting
   execution.
*/
extern void exit(int __status) __ATTR_NORETURN__;

/**
   The malloc() function allocates \c size bytes of memory.
   If malloc() fails, a NULL pointer is returned.

   Note that malloc() does \e not initialize the returned memory to
   zero bytes.

   See the chapter about \ref malloc "malloc() usage" for implementation
   details.
*/
extern void *malloc(size_t __size) __ATTR_MALLOC__;

/**
   The free() function causes the allocated memory referenced by \c
   ptr to be made available for future allocations.  If \c ptr is
   NULL, no action occurs.
*/
extern void free(void *__ptr);

/**
   \c malloc() \ref malloc_tunables "tunable".
*/
extern size_t __malloc_margin;

/**
   \c malloc() \ref malloc_tunables "tunable".
*/
extern char *__malloc_heap_start;

/**
   \c malloc() \ref malloc_tunables "tunable".
*/
extern char *__malloc_heap_end;

/**
   Allocate \c nele elements of \c size each.  Identical to calling
   \c malloc() using <tt>nele * size</tt> as argument, except the
   allocated memory will be cleared to zero.
*/
extern void *calloc(size_t __nele, size_t __size) __ATTR_MALLOC__;

/**
   The realloc() function tries to change the size of the region
   allocated at \c ptr to the new \c size value.  It returns a
   pointer to the new region.  The returned pointer might be the
   same as the old pointer, or a pointer to a completely different
   region.
```

```
   The contents of the returned region up to either the old or the new
   size value (whatever is less) will be identical to the contents of
   the old region, even in case a new region had to be allocated.

   It is acceptable to pass \c ptr as NULL, in which case realloc()
   will behave identical to malloc().

   If the new memory cannot be allocated, realloc() returns NULL, and
   the region at \c ptr will not be changed.
*/
extern void *realloc(void *__ptr, size_t __size) __ATTR_MALLOC__;

extern double strtod(const char *__nptr, char **__endptr);

extern double atof(const char *__nptr);

/** Highest number that can be generated by rand(). */
#define RAND_MAX 0x7FFF

/**
     The rand() function computes a sequence of pseudo-random integers in the
     range of 0 to \c RAND_MAX (as defined by the header file <stdlib.h>).

     The srand() function sets its argument \c seed as the seed for a new
     sequence of pseudo-random numbers to be returned by rand().  These
     sequences are repeatable by calling srand() with the same seed value.

     If no seed value is provided, the functions are automatically seeded with
     a value of 1.

     In compliance with the C standard, these functions operate on
     \c int arguments.  Since the underlying algorithm already uses
     32-bit calculations, this causes a loss of precision.  See
     \c random() for an alternate set of functions that retains full
     32-bit precision.
*/
extern int rand(void);
/**
   Pseudo-random number generator seeding; see rand().
*/
extern void srand(unsigned int __seed);

/**
   Variant of rand() that stores the context in the user-supplied
   variable located at \c ctx instead of a static library variable
   so the function becomes re-entrant.
*/
extern int rand_r(unsigned long *__ctx);
/*@}*/

/*@{*/
/** \name Non-standard (i.e. non-ISO C) functions.
 \ingroup avr_stdlib
*/
/**
   \brief Convert an integer to a string.

   The function itoa() converts the integer value from \c val into an
   ASCII representation that will be stored under \c s.  The caller
   is responsible for providing sufficient storage in \c s.

   \note The minimal size of the buffer \c s depends on the choice of
   radix. For example, if the radix is 2 (binary), you need to supply a buffer
   with a minimal length of 8 * sizeof (int) + 1 characters, i.e. one
   character for each bit plus one for the string terminator. Using a larger
   radix will require a smaller minimal buffer size.
```

```
   \warning If the buffer is too small, you risk a buffer overflow.

   Conversion is done using the \c radix as base, which may be a
   number between 2 (binary conversion) and up to 36.  If \c radix
   is greater than 10, the next digit after \c '9' will be the letter
   \c 'a'.

    If radix is 10 and val is negative, a minus sign will be prepended.

   The itoa() function returns the pointer passed as \c s.
*/
#ifdef  __DOXYGEN__
extern char *itoa(int val, char *s, int radix);
#else
extern __inline__ __ATTR_GNU_INLINE__
char *itoa (int __val, char *__s, int __radix)
{
    if (!__builtin_constant_p (__radix)) {
  extern char *__itoa (int, char *, int);
  return __itoa (__val, __s, __radix);
    } else if (__radix < 2 || __radix > 36) {
  *__s = 0;
  return __s;
    } else {
  extern char *__itoa_ncheck (int, char *, unsigned char);
  return __itoa_ncheck (__val, __s, __radix);
    }
}
#endif

/**
 \ingroup avr_stdlib

   \brief Convert a long integer to a string.

   The function ltoa() converts the long integer value from \c val into an
   ASCII representation that will be stored under \c s.  The caller
   is responsible for providing sufficient storage in \c s.

   \note The minimal size of the buffer \c s depends on the choice of
   radix. For example, if the radix is 2 (binary), you need to supply a buffer
   with a minimal length of 8 * sizeof (long int) + 1 characters, i.e. one
   character for each bit plus one for the string terminator. Using a larger
   radix will require a smaller minimal buffer size.

   \warning If the buffer is too small, you risk a buffer overflow.

   Conversion is done using the \c radix as base, which may be a
   number between 2 (binary conversion) and up to 36.  If \c radix
   is greater than 10, the next digit after \c '9' will be the letter
   \c 'a'.

   If radix is 10 and val is negative, a minus sign will be prepended.

   The ltoa() function returns the pointer passed as \c s.
*/
#ifdef  __DOXYGEN__
extern char *ltoa(long val, char *s, int radix);
#else
extern __inline__ __ATTR_GNU_INLINE__
char *ltoa (long __val, char *__s, int __radix)
{
    if (!__builtin_constant_p (__radix)) {
  extern char *__ltoa (long, char *, int);
  return __ltoa (__val, __s, __radix);
    } else if (__radix < 2 || __radix > 36) {
  *__s = 0;
```

```
  return __s;
    } else {
  extern char *__ltoa_ncheck (long, char *, unsigned char);
  return __ltoa_ncheck (__val, __s, __radix);
    }
}
#endif

/**
 \ingroup avr_stdlib

   \brief Convert an unsigned integer to a string.

   The function utoa() converts the unsigned integer value from \c val into an
   ASCII representation that will be stored under \c s.  The caller
   is responsible for providing sufficient storage in \c s.

   \note The minimal size of the buffer \c s depends on the choice of
   radix. For example, if the radix is 2 (binary), you need to supply a buffer
   with a minimal length of 8 * sizeof (unsigned int) + 1 characters, i.e. one
   character for each bit plus one for the string terminator. Using a larger
   radix will require a smaller minimal buffer size.

   \warning If the buffer is too small, you risk a buffer overflow.

   Conversion is done using the \c radix as base, which may be a
   number between 2 (binary conversion) and up to 36.  If \c radix
   is greater than 10, the next digit after \c '9' will be the letter
   \c 'a'.

   The utoa() function returns the pointer passed as \c s.
*/
#ifdef  __DOXYGEN__
extern char *utoa(unsigned int val, char *s, int radix);
#else
extern __inline__ __ATTR_GNU_INLINE__
char *utoa (unsigned int __val, char *__s, int __radix)
{
    if (!__builtin_constant_p (__radix)) {
  extern char *__utoa (unsigned int, char *, int);
  return __utoa (__val, __s, __radix);
    } else if (__radix < 2 || __radix > 36) {
  *__s = 0;
  return __s;
    } else {
  extern char *__utoa_ncheck (unsigned int, char *, unsigned char);
  return __utoa_ncheck (__val, __s, __radix);
    }
}
#endif

/**
 \ingroup avr_stdlib
   \brief Convert an unsigned long integer to a string.

   The function ultoa() converts the unsigned long integer value from
   \c val into an ASCII representation that will be stored under \c s.
   The caller is responsible for providing sufficient storage in \c s.

   \note The minimal size of the buffer \c s depends on the choice of
   radix. For example, if the radix is 2 (binary), you need to supply a buffer
   with a minimal length of 8 * sizeof (unsigned long int) + 1 characters,
   i.e. one character for each bit plus one for the string terminator. Using a
   larger radix will require a smaller minimal buffer size.

   \warning If the buffer is too small, you risk a buffer overflow.
```

```
   Conversion is done using the \c radix as base, which may be a
   number between 2 (binary conversion) and up to 36.  If \c radix
   is greater than 10, the next digit after \c '9' will be the letter
   \c 'a'.

   The ultoa() function returns the pointer passed as \c s.
*/
#ifdef  __DOXYGEN__
extern char *ultoa(unsigned long val, char *s, int radix);
#else
extern __inline__ __ATTR_GNU_INLINE__
char *ultoa (unsigned long __val, char *__s, int __radix)
{
    if (!__builtin_constant_p (__radix)) {
  extern char *__ultoa (unsigned long, char *, int);
  return __ultoa (__val, __s, __radix);
    } else if (__radix < 2 || __radix > 36) {
  *__s = 0;
  return __s;
    } else {
  extern char *__ultoa_ncheck (unsigned long, char *, unsigned char);
  return __ultoa_ncheck (__val, __s, __radix);
    }
}
#endif

/**  \ingroup avr_stdlib
Highest number that can be generated by random(). */
#define RANDOM_MAX 0x7FFFFFFF

/**
 \ingroup avr_stdlib
     The random() function computes a sequence of pseudo-random integers in the
     range of 0 to \c RANDOM_MAX (as defined by the header file <stdlib.h>).

     The srandom() function sets its argument \c seed as the seed for a new
     sequence of pseudo-random numbers to be returned by rand().  These
     sequences are repeatable by calling srandom() with the same seed value.

     If no seed value is provided, the functions are automatically seeded with
     a value of 1.
*/
extern long random(void);
/**
 \ingroup avr_stdlib
   Pseudo-random number generator seeding; see random().
*/
extern void srandom(unsigned long __seed);

/**
 \ingroup avr_stdlib
   Variant of random() that stores the context in the user-supplied
   variable located at \c ctx instead of a static library variable
   so the function becomes re-entrant.
*/
extern long random_r(unsigned long *__ctx);
#endif /* __ASSEMBLER */
/*@}*/

/*@{*/
/** \name Conversion functions for double arguments.
 \ingroup avr_stdlib
 Note that these functions are not located in the default library,
 <tt>libc.a</tt>, but in the mathematical library, <tt>libm.a</tt>.
 So when linking the application, the \c -lm option needs to be
 specified.
*/
```

```
/** \ingroup avr_stdlib
    Bit value that can be passed in \c flags to dtostre(). */
#define DTOSTR_ALWAYS_SIGN 0x01        /* put '+' or ' ' for positives */
/** \ingroup avr_stdlib
    Bit value that can be passed in \c flags to dtostre(). */
#define DTOSTR_PLUS_SIGN   0x02        /* put '+' rather than ' ' */
/** \ingroup avr_stdlib
    Bit value that can be passed in \c flags to dtostre(). */
#define DTOSTR_UPPERCASE   0x04        /* put 'E' rather 'e' */

#ifndef __ASSEMBLER__

/**
   \ingroup avr_stdlib
   The dtostre() function converts the double value passed in \c val into
   an ASCII representation that will be stored under \c s.  The caller
   is responsible for providing sufficient storage in \c s.

   Conversion is done in the format \c "[-]d.ddde±dd" where there is
   one digit before the decimal-point character and the number of
   digits after it is equal to the precision \c prec; if the precision
   is zero, no decimal-point character appears.  If \c flags has the
   DTOSTRE_UPPERCASE bit set, the letter \c 'E' (rather than \c 'e' ) will be
   used to introduce the exponent.  The exponent always contains two
   digits; if the value is zero, the exponent is \c "00".

   If \c flags has the DTOSTRE_ALWAYS_SIGN bit set, a space character
   will be placed into the leading position for positive numbers.

   If \c flags has the DTOSTRE_PLUS_SIGN bit set, a plus sign will be
   used instead of a space character in this case.

   The dtostre() function returns the pointer to the converted string \c s.
*/
extern char *dtostre(double __val, char *__s, unsigned char __prec,
         unsigned char __flags);

/**
   \ingroup avr_stdlib
   The dtostrf() function converts the double value passed in \c val into
   an ASCII representationthat will be stored under \c s.  The caller
   is responsible for providing sufficient storage in \c s.

   Conversion is done in the format \c "[-]d.ddd".  The minimum field
   width of the output string (including the \c '.' and the possible
   sign for negative values) is given in \c width, and \c prec determines
   the number of digits after the decimal sign. \c width is signed value,
   negative for left adjustment.

   The dtostrf() function returns the pointer to the converted string \c s.
*/
extern char *dtostrf(double __val, signed char __width,
                     unsigned char __prec, char *__s);

/**
   \ingroup avr_stdlib
    Successful termination for exit(); evaluates to 0.
*/
#define EXIT_SUCCESS 0

/**
   \ingroup avr_stdlib
    Unsuccessful termination for exit(); evaluates to a non-zero value.
*/
#define EXIT_FAILURE 1

/*@}*/
```

```
#if 0  /* not yet implemented */
extern int atexit(void (*)(void));
#endif

#ifdef __cplusplus
}
#endif

#endif /* __ASSEMBLER */

#endif /* _STDLIB_H_ */
```