Goto sanos source index

```
//
// string.c
//
// String routines
//
// Copyright (C) 2002 Michael Ringgaard. All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions
// are met:
//
// 1. Redistributions of source code must retain the above copyright
//    notice, this list of conditions and the following disclaimer.
// 2. 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.
// 3. Neither the name of the project nor the names of its 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.
//

#include <os.h>
#include <string.h>

#ifndef KERNEL
#include <ctype.h>
#endif

char *strncpy(char *dest, const char *source, size_t n) {
  char *start = dest;

  while (n && (*dest++ = *source++)) n--;
  if (n) while (--n) *dest++ = '\0';
  return start;
}

int strncmp(const char *s1, const char *s2, size_t n) {
  if (!n) return 0;

  while (--n && *s1 && *s1 == *s2) {
    s1++;
    s2++;
  }

  return *(unsigned char *) s1 - *(unsigned char *) s2;
}

int stricmp(const char *s1, const char *s2) {
  char f, l;

  do {
    f = ((*s1 <= 'Z') && (*s1 >= 'A')) ? *s1 + 'a' - 'A' : *s1;
    l = ((*s2 <= 'Z') && (*s2 >= 'A')) ? *s2 + 'a' - 'A' : *s2;
    s1++;
    s2++;
  } while ((f) && (f == l));
```

```c
  return (int) (f - l);
}

int strnicmp(const char *s1, const char *s2, size_t n) {
  int f, l;

  do {
    if (((f = (unsigned char)(*(s1++))) >= 'A') && (f <= 'Z')) f -= 'A' - 'a';
    if (((l = (unsigned char)(*(s2++))) >= 'A') && (l <= 'Z')) l -= 'A' - 'a';
  } while (--n && f && (f == l));

  return f - l;
}

int strcasecmp(const char *s1, const char *s2) {
  return stricmp(s1, s2);
}

int strncasecmp(const char *s1, const char *s2, size_t n) {
  return strnicmp(s1, s2, n);
}

char *strchr(const char *s, int ch) {
  while (*s && *s != (char) ch) s++;
  if (*s == (char) ch) return (char *) s;
  return NULL;
}

char *strrchr(const char *s, int ch) {
  char *start = (char *) s;

  while (*s++);
  while (--s != start && *s != (char) ch);
  if (*s == (char) ch) return (char *) s;

  return NULL;
}

char *strstr(const char *str1, const char *str2) {
  char *cp = (char *) str1;
  char *s1, *s2;

  if (!*str2) return (char *) str1;

  while (*cp) {
    s1 = cp;
    s2 = (char *) str2;

    while (*s1 && *s2 && !(*s1 - *s2)) s1++, s2++;
    if (!*s2) return cp;
    cp++;
  }

  return NULL;
}

size_t strspn(const char *string, const char *control) {
  const unsigned char *str = string;
  const unsigned char *ctrl = control;

  unsigned char map[32];
  int n;

  // Clear out bit map
  for (n = 0; n < 32; n++) map[n] = 0;

  // Set bits in control map
  while (*ctrl) {
    map[*ctrl >> 3] |= (1 << (*ctrl & 7));
    ctrl++;
  }
  // 1st char NOT in control map stops search
  if (*str) {
    n = 0;
    while (map[*str >> 3] & (1 << (*str & 7))) {
      n++;
      str++;
    }

    return n;
  }

  return 0;
}

size_t strcspn(const char *string, const char *control) {
  const unsigned char *str = string;
  const unsigned char *ctrl = control;

  unsigned char map[32];
  int n;

  // Clear out bit map
  for (n = 0; n < 32; n++) map[n] = 0;

  // Set bits in control map
  while (*ctrl) {
    map[*ctrl >> 3] |= (1 << (*ctrl & 7));
    ctrl++;
  }

  // 1st char in control map stops search
  n = 0;
  map[0] |= 1;
  while (!(map[*str >> 3] & (1 << (*str & 7)))) {
    n++;
    str++;
  }
  return n;
}

char *strpbrk(const char *string, const char *control) {
  const unsigned char *str = string;
  const unsigned char *ctrl = control;

  unsigned char map[32];
  int n;

  // Clear out bit map
  for (n = 0; n < 32; n++) map[n] = 0;

  // Set bits in control map
  while (*ctrl) {
    map[*ctrl >> 3] |= (1 << (*ctrl & 7));
    ctrl++;
  }

  // 1st char in control map stops search
  while (*str) {
    if (map[*str >> 3] & (1 << (*str & 7))) return (char *) str;
    str++;
  }

  return NULL;
}

void *memmove(void *dst, const void *src, size_t n) {
  void * ret = dst;

  if (dst <= src || (char *) dst >= ((char *) src + n)) {
    // Non-overlapping buffers; copy from lower addresses to higher addresses
    while (n--) {
      *(char *) dst = *(char *) src;
      dst = (char *) dst + 1;
      src = (char *) src + 1;
    }
  } else {
    // Overlapping buffers; copy from higher addresses to lower addresses
    dst = (char *) dst + n - 1;
    src = (char *) src + n - 1;

    while (n--) {
      *(char *) dst = *(char *) src;
      dst = (char *) dst - 1;
      src = (char *) src - 1;
    }
  }

  return ret;
}

void *memchr(const void *buf, int ch, size_t n) {
  while (n && (*(unsigned char *) buf != (unsigned char) ch)) {
    buf = (unsigned char *) buf + 1;
    n--;
  }

  return (n ? (void *) buf : NULL);
}

#ifndef KERNEL

char *strdup(const char *s) {
  char *t;
  int len;

  if (!s) return NULL;
  len = strlen(s);
  t = (char *) malloc(len + 1);
  memcpy(t, s, len + 1);
  return t;
}

char *_lstrdup(const char *s) {
  char *t;
  int len;

  if (!s) return NULL;
  len = strlen(s);
  t = (char *) _lmalloc(len + 1);
  memcpy(t, s, len + 1);
  return t;
}

char *strlwr(char *s) {
  char *p = s;

  while (*p) {
    *p = (char) tolower(*p);
    p++;
  }

  return s;
}

char *strupr(char *s) {
  char *p = s;

  while (*p) {
    *p = (char) toupper(*p);
    p++;
  }

  return s;
```

```c
}

#endif

char *strncat(char *s1, const char *s2, size_t n) {
  char *start = s1;

  while (*s1++);
  s1--;

  while (n--) {
    if (!(*s1++ = *s2++)) return start;
  }

  *s1 = '\0';
  return start;
}

char *strnset(char *s, int c, size_t n) {
  char *start = s;
  while (n-- && *s) *s++ = (char) c;
  return s;
}

char *strrev(char *s) {
  char *start = s;
  char *left = s;
  char ch;

  while (*s++);
  s -= 2;

  while (left < s) {
    ch = *left;
    *left++ = *s;
    *s-- = ch;
  }

  return start;
}

char *strtok_r(char *string, const char *control, char **lasts) {
  unsigned char *str;
  const unsigned char *ctrl = control;

  unsigned char map[32];
  int n;

  // Clear control map
  for (n = 0; n < 32; n++) map[n] = 0;

  // Set bits in delimiter table
  do { map[*ctrl >> 3] |= (1 << (*ctrl & 7)); } while (*ctrl++);

  // Initialize str. If string is NULL, set str to the saved
  // pointer (i.e., continue breaking tokens out of the string
  // from the last strtok call)
  if (string) {
    str = string;
  } else {
    str = *lasts;
  }

  // Find beginning of token (skip over leading delimiters). Note that
  // there is no token iff this loop sets str to point to the terminal
  // null (*str == '\0')

  while ((map[*str >> 3] & (1 << (*str & 7))) && *str) str++;

  string = str;

  // Find the end of the token. If it is not the end of the string,
  // put a null there
  for ( ; *str ; str++) {
    if (map[*str >> 3] & (1 << (*str & 7))) {
      *str++ = '\0';
      break;
    }
  }

  // Update nexttoken
  *lasts = str;

  // Determine if a token has been found
  if (string == (char *) str) {
    return NULL;
  } else {
    return string;
  }
}

#ifndef KERNEL

char *strtok(char *string, const char *control) {
  return strtok_r(string, control, &gettib()->nexttoken);
}

#endif

char *strsep(char **stringp, const char *delim) {
  char *s;
  const char *d;
  char *start;
  int c;

  start = *stringp;
  if (!start) return NULL;
  s = start;
  while (c = *s++) {
    for (d = delim; *d; d++) {
      if (c == *d) {
        s[-1] = 0;
        *stringp = s;
        return start;
      }
    }
  }
  *stringp = NULL;
  return start;
}

/////////////////////////////////////////////////////////////////////
//
// intrinsic functions
//

#pragma function(memset)
#pragma function(memcmp)
#pragma function(memcpy)

#pragma function(strcpy)
#pragma function(strlen)
#pragma function(strcat)
#pragma function(strcmp)
#pragma function(strset)

void *memset(void *p, int c, size_t n) {
  char *pb = (char *) p;
  char *pbend = pb + n;
  while (pb != pbend) *pb++ = c;
  return p;
}

int memcmp(const void *dst, const void *src, size_t n) {
  if (!n) return 0;

  while (--n && *(char *) dst == *(char *) src) {
    dst = (char *) dst + 1;
    src = (char *) src + 1;
  }

  return *((unsigned char *) dst) - *((unsigned char *) src);
}

#ifdef __i386__
void *memcpy(void *dst, const void *src, size_t n) {
  __asm {
    push  esi
    push  edi
    mov    esi,src
    mov    edi,dst
    mov    ecx,n

    mov    eax,esi
    or     eax,edi
    or     eax,n
    and    eax, 3
    jz     fast_copy

    rep    movsb
    jmp    copy_done

fast_copy:
    shr    ecx,2
    rep    movsd

copy_done:
    mov    eax,dst
    pop    edi
    pop    esi
  }
}
#else
void *memcpy(void *dst, const void *src, size_t n) {
  char *s = (char *) src;
  char *end = s + n;
  char *d = (char *) dst;
  if ((((unsigned int) s) | ((unsigned int) d) | n) && sizeof(unsigned int) - 1) {
    while (s != end) *d++ = *s++;
  } else {
    while (s != end) *((unsigned int *) d)++ = *((unsigned int *) s)++;
  }

  return dst;
}
#endif

void *memccpy(void *dst, const void *src, int c, size_t n) {
  while (n && (*((char *) (dst = (char *) dst + 1) - 1) =
          *((char *)(src = (char *) src + 1) - 1)) != (char) c) {
    n--;
  }

  return n ? dst : NULL;
}

#ifndef KERNEL

int memicmp(const void *buf1, const void *buf2, size_t n) {
  int f = 0, l = 0;
  const unsigned char *dst = buf1, *src = buf2;

  while (n-- && f == l) {
    f = tolower(*dst++);
    l = tolower(*src++);
  }
  return f - l;
}

#endif

char *strcpy(char *dst, const char *src) {
  char *cp = dst;
  while (*cp++ = *src++);
  return dst;
}

size_t strlen(const char *s) {
  const char *eos = s;
  while (*eos++);
  return (int) (eos - s - 1);
}

int strcmp(const char *s1, const char *s2) {
  int ret = 0;
  while (!(ret = *(unsigned char *) s1 - *(unsigned char *) s2) && *s2) ++s1, ++s2;

  if (ret < 0) {
    ret = -1;
  } else if (ret > 0) {
    ret = 1 ;
  }

  return ret;
}

char *strcat(char *dst, const char *src) {
  char *cp = dst;
  while (*cp) cp++;
  while (*cp++ = *src++);
  return dst;
}

char *strset(char *s, int c) {
  char *start = s;
  while (*s) *s++ = (char) c;
  return start;
}
```