95.1 os32: file isolati della directory «lib/» ... 753
95.1.1 lib/NULL.h ... 753
95.1.2 lib/SEEK.h ... 753
95.1.3 lib/assert.h ... 753
95.1.4 lib/clock_t.h ... 754
95.1.5 lib/ctype.h ... 754
95.1.6 lib/limits.h ... 754
95.1.7 lib/ptrdiff_t.h ... 755
95.1.8 lib/restrict.h ... 755
95.1.9 lib/size_t.h ... 756
95.1.10 lib/stdarg.h ... 756
95.1.11 lib/stdbool.h ... 756
95.1.12 lib/stddef.h ... 756
95.1.13 lib/stdint.h ... 756
95.1.14 lib/time_t.h ... 758
95.1.15 lib/wchar_t.h ... 758
95.2 os32: «lib/_gcc.h» ... 758
95.2.1 lib/_gcc/__divdi3.c ... 759
95.2.2 lib/_gcc/__moddi3.c ... 759
95.2.3 lib/_gcc/__udivdi3.c ... 759
95.2.4 lib/_gcc/__umoddi3.c ... 759
95.2.5 lib/_gcc/_lldiv.c ... 759
95.2.6 lib/_gcc/_ulldiv.c ... 760
95.3 os32: «lib/arpa/inet.h» ... 761
95.3.1 lib/arpa/inet/htonl.c ... 761
95.3.2 lib/arpa/inet/htons.c ... 762
95.3.3 lib/arpa/inet/inet_ntop.c ... 762
95.3.4 lib/arpa/inet/inet_pton.c ... 762
95.3.5 lib/arpa/inet/ntohl.c ... 764
95.3.6 lib/arpa/inet/ntohs.c ... 764
95.4 os32: «lib/dirent.h» ... 764
95.4.1 lib/dirent/DIR.c ... 765
95.4.2 lib/dirent/closedir.c ... 765
95.4.3 lib/dirent/opendir.c ... 766
95.4.4 lib/dirent/readdir.c ... 767
95.4.5 lib/dirent/rewinddir.c ... 768
95.5 os32: «lib/errno.h» ... 768
95.5.1 lib/errno/errno.c ... 773
95.6 os32: «lib/fcntl.h» ... 773
95.6.1 lib/fcntl/creat.c ... 774
95.6.2 lib/fcntl/fcntl.c ... 775
95.6.3 lib/fcntl/open.c ... 775
95.7 os32: «lib/grp.h» ... 776
95.7.1 lib/grp/grent.c ... 776
95.8 os32: «lib/inttypes.h» ... 777
95.8.1 lib/inttypes/imaxabs.c ... 781
95.8.2 lib/inttypes/imaxdiv.c ... 781
95.9 os32: «lib/libgen.h» ... 781
95.9.1 lib/libgen/basename.c ... 781
95.9.2 lib/libgen/dirname.c ... 782
95.10 os32: «lib/netinet/icmp.h» ... 783
95.11 os32: «lib/netinet/in.h» ... 785

95.12 os32: «lib/netinet/ip.h» 786
95.13 os32: «lib/netinet/tcp.h» 787
95.14 os32: «lib/netinet/udp.h» 788
95.15 os32: «lib/pwd.h» 788
95.15.1 lib/pwd/pwent.c 789
95.16 os32: «lib/setjmp.h» 790
95.16.1 lib/setjmp/longjmp.c 791
95.16.2 lib/setjmp/setjmp.s 791
95.17 os32: «lib/signal.h» 792
95.17.1 lib/signal/_sighandler_wrapper.s 793
95.17.2 lib/signal/kill.c 794
95.17.3 lib/signal/signal.c 794
95.18 os32: «lib/stdio.h» 794
95.18.1 lib/stdio/FILE.c 796
95.18.2 lib/stdio/clearerr.c 797
95.18.3 lib/stdio/fclose.c 797
95.18.4 lib/stdio/feof.c 797
95.18.5 lib/stdio/ferror.c 797
95.18.6 lib/stdio/fflush.c 798
95.18.7 lib/stdio/fgetc.c 798
95.18.8 lib/stdio/fgetpos.c 798
95.18.9 lib/stdio/fgets.c 798
95.18.10 lib/stdio/fileno.c 799
95.18.11 lib/stdio/fopen.c 799
95.18.12 lib/stdio/fprintf.c 800
95.18.13 lib/stdio/fputc.c 800
95.18.14 lib/stdio/fputs.c 801
95.18.15 lib/stdio/fread.c 801
95.18.16 lib/stdio/freopen.c 801
95.18.17 lib/stdio/fscanf.c 802
95.18.18 lib/stdio/fseek.c 802
95.18.19 lib/stdio/fseeko.c 802
95.18.20 lib/stdio/fsetpos.c 802
95.18.21 lib/stdio/ftell.c 803
95.18.22 lib/stdio/ftello.c 803
95.18.23 lib/stdio/fwrite.c 803
95.18.24 lib/stdio/getchar.c 803
95.18.25 lib/stdio/gets.c 804
95.18.26 lib/stdio/perror.c 804
95.18.27 lib/stdio/printf.c 805
95.18.28 lib/stdio/putchar.c 805
95.18.29 lib/stdio/puts.c 805
95.18.30 lib/stdio/rewind.c 806
95.18.31 lib/stdio/scanf.c 806
95.18.32 lib/stdio/setbuf.c 806
95.18.33 lib/stdio/setvbuf.c 806
95.18.34 lib/stdio/snprintf.c 806
95.18.35 lib/stdio/sprintf.c 806
95.18.36 lib/stdio/sscanf.c 807
95.18.37 lib/stdio/vfprintf.c 807
95.18.38 lib/stdio/vfscanf.c 807
95.18.39 lib/stdio/vfsscanf.c 807
95.18.40 lib/stdio/vprintf.c 827
95.18.41 lib/stdio/vscanf.c 827
95.18.42 lib/stdio/vsnprintf.c 828
95.18.43 lib/stdio/vsprintf.c 843

95.18.44 lib/stdio/vsscanf.c 844
95.19 os32: «lib/stdlib.h» 844
95.19.1 lib/stdlib/_Exit.c 846
95.19.2 lib/stdlib/abort.c 846
95.19.3 lib/stdlib/abs.c 846
95.19.4 lib/stdlib/atexit.c 847
95.19.5 lib/stdlib/atoi.c 847
95.19.6 lib/stdlib/atol.c 848
95.19.7 lib/stdlib/div.c 848
95.19.8 lib/stdlib/environment.c 848
95.19.9 lib/stdlib/exit.c 849
95.19.10 lib/stdlib/getenv.c 850
95.19.11 lib/stdlib/labs.c 851
95.19.12 lib/stdlib/ldiv.c 851
95.19.13 lib/stdlib/llabs.c 851
95.19.14 lib/stdlib/lldiv.c 851
95.19.15 lib/stdlib/putenv.c 851
95.19.16 lib/stdlib/qsort.c 853
95.19.17 lib/stdlib/rand.c 854
95.19.18 lib/stdlib/setenv.c 855
95.19.19 lib/stdlib/strtol.c 856
95.19.20 lib/stdlib/strtoul.c 859
95.19.21 lib/stdlib/unsetenv.c 859
95.19.22 lib/stdlib_alloc/_alloc_list.c 860
95.19.23 lib/stdlib_alloc/free.c 861
95.19.24 lib/stdlib_alloc/malloc.c 862
95.19.25 lib/stdlib_alloc/realloc.c 865
95.20 os32: «lib/string.h» 866
95.20.1 lib/string/memccpy.c 867
95.20.2 lib/string/memchr.c 868
95.20.3 lib/string/memcmp.c 868
95.20.4 lib/string/memcpy.c 868
95.20.5 lib/string/memmove.c 868
95.20.6 lib/string/memset.c 869
95.20.7 lib/string/strcat.c 869
95.20.8 lib/string/strchr.c 869
95.20.9 lib/string/strcmp.c 870
95.20.10 lib/string/strcoll.c 870
95.20.11 lib/string/strcpy.c 870
95.20.12 lib/string/strcspn.c 870
95.20.13 lib/string/strdup.c 871
95.20.14 lib/string/strerror.c 871
95.20.15 lib/string/strlen.c 873
95.20.16 lib/string/strncat.c 873
95.20.17 lib/string/strncmp.c 873
95.20.18 lib/string/strncpy.c 873
95.20.19 lib/string/strpbrk.c 874
95.20.20 lib/string/strrchr.c 874
95.20.21 lib/string/strspn.c 874
95.20.22 lib/string/strstr.c 875
95.20.23 lib/string/strtok.c 875
95.20.24 lib/string/strxfrm.c 877
95.21 os32: «lib/sys/os32.h» 877
95.21.1 lib/sys/os32/input_line.c 885
95.21.2 lib/sys/os32/ipconfig.c 887
95.21.3 lib/sys/os32/mount.c 888
95.21.4 lib/sys/os32/namep.c 888

95.21.5 lib/sys/os32/routeadd.c 890
95.21.6 lib/sys/os32/routedel.c 891
95.21.7 lib/sys/os32/sys.s 891
95.21.8 lib/sys/os32/umount.c 891
95.21.9 lib/sys/os32/z_perror.c 892
95.21.10 lib/sys/os32/z_printf.c 892
95.21.11 lib/sys/os32/z_vprintf.c 892
95.22 os32: «lib/sys/sa_family_t.h» 893
95.23 os32: «lib/sys/socket.h» 893
95.23.1 lib/sys/socket/accept.c 894
95.23.2 lib/sys/socket/bind.c 895
95.23.3 lib/sys/socket/connect.c 895
95.23.4 lib/sys/socket/listen.c 896
95.23.5 lib/sys/socket/recvfrom.c 896
95.23.6 lib/sys/socket/send.c 898
95.23.7 lib/sys/socket/socket.c 899
95.24 os32: «lib/sys/socklen_t.h» 899
95.25 os32: «lib/sys/stat.h» 899
95.25.1 lib/sys/stat/chmod.c 901
95.25.2 lib/sys/stat/fchmod.c 901
95.25.3 lib/sys/stat/fstat.c 902
95.25.4 lib/sys/stat/mkdir.c 902
95.25.5 lib/sys/stat/mknod.c 903
95.25.6 lib/sys/stat/stat.c 903
95.25.7 lib/sys/stat/umask.c 903
95.26 os32: «lib/sys/types.h» 904
95.26.1 lib/sys/types/major.c 904
95.26.2 lib/sys/types/makedev.c 904
95.26.3 lib/sys/types/minor.c 904
95.27 os32: «lib/sys/wait.h» 905
95.27.1 lib/sys/wait/wait.c 905
95.28 os32: «lib/termios.h» 905
95.28.1 lib/termios/tcgetattr.c 906
95.28.2 lib/termios/tcsetattr.c 907
95.29 os32: «lib/time.h» 907
95.29.1 lib/time/asctime.c 908
95.29.2 lib/time/clock.c 909
95.29.3 lib/time/gmtime.c 909
95.29.4 lib/time/mktime.c 911
95.29.5 lib/time/stime.c 913
95.29.6 lib/time/time.c 913
95.30 os32: «lib/unistd.h» 913
95.30.1 lib/unistd/_exit.c 915
95.30.2 lib/unistd/access.c 916
95.30.3 lib/unistd/brk.c 916
95.30.4 lib/unistd/chdir.c 917
95.30.5 lib/unistd/chown.c 917
95.30.6 lib/unistd/close.c 917
95.30.7 lib/unistd/dup.c 918
95.30.8 lib/unistd/dup2.c 918
95.30.9 lib/unistd/environ.c 918
95.30.10 lib/unistd/execl.c 918
95.30.11 lib/unistd/execle.c 919
95.30.12 lib/unistd/execlp.c 919
95.30.13 lib/unistd/execv.c 920

95.30.14 lib/unistd/execve.c 920
95.30.15 lib/unistd/execvp.c 921
95.30.16 lib/unistd/fchdir.c 922
95.30.17 lib/unistd/fchown.c 922
95.30.18 lib/unistd/fork.c 922
95.30.19 lib/unistd/getcwd.c 923
95.30.20 lib/unistd/getegid.c 923
95.30.21 lib/unistd/geteuid.c 924
95.30.22 lib/unistd/getgid.c 924
95.30.23 lib/unistd/getopt.c 924
95.30.24 lib/unistd/getpgrp.c 927
95.30.25 lib/unistd/getpid.c 927
95.30.26 lib/unistd/getppid.c 928
95.30.27 lib/unistd/getuid.c 928
95.30.28 lib/unistd/isatty.c 928
95.30.29 lib/unistd/link.c 929
95.30.30 lib/unistd/lseek.c 929
95.30.31 lib/unistd/pipe.c 929
95.30.32 lib/unistd/read.c 930
95.30.33 lib/unistd/rmdir.c 931
95.30.34 lib/unistd/sbrk.c 931
95.30.35 lib/unistd/setegid.c 932
95.30.36 lib/unistd/seteuid.c 932
95.30.37 lib/unistd/setgid.c 932
95.30.38 lib/unistd/setpgrp.c 932
95.30.39 lib/unistd/setuid.c 933
95.30.40 lib/unistd/sleep.c 933
95.30.41 lib/unistd/ttyname.c 933
95.30.42 lib/unistd/unlink.c 934
95.30.43 lib/unistd/write.c 934
95.31 os32: «lib/utime.h» 935
95.31.1 lib/utime/utime.c 935

95.1 os32: file isolati della directory «lib/»

95.1.1 lib/NULL.h

Si veda la sezione 91.3.

```
3150001 #ifndef _NULL_H
3150002 #define _NULL_H      1
3150003 //-------------------------------------------------------------
3150004 #define NULL ((void *) 0)
3150005 //-------------------------------------------------------------
3150006 #endif
```

95.1.2 lib/SEEK.h

Si veda la sezione 91.3.

```
3160001 #ifndef _SEEK_H
3160002 #define _SEEK_H      1
3160003 //-------------------------------------------------------------
3160004 // These values are used inside 'stdio.h' and
3160005 // 'unistd.h'.
3160006 //-------------------------------------------------------------
3160007 #define SEEK_SET        0       // From the start.
3160008 #define SEEK_CUR        1       // From current
3160009                                 // position.
3160010 #define SEEK_END        2       // From the end.
3160011 //-------------------------------------------------------------
3160012 #endif
```

95.1.3 lib/assert.h

Si veda la sezione 88.6.

```
3170001 #ifndef _ASSERT_H
3170002 #define _ASSERT_H       1
3170003 //-------------------------------------------------------------
```

```
#include <stdio.h>
//-----------------------------------------------------------
#ifdef NDEBUG
#define assert(ignore) ((void)0)
#else
#define assert(ASSERTION)                                \
    ({if ((ASSERTION)==0)                                \
       fprintf (stderr,                                  \
                "Assertion failed: " # ASSERTION         \
                ", function %s, file %s, line %u.\n", \
                __func__, __FILE__, __LINE__);})
#endif
//-----------------------------------------------------------
#endif
```

95.1.4 lib/clock_t.h

Si veda la sezione 91.3.

```
#ifndef _CLOCK_T_H
#define _CLOCK_T_H       1
//-----------------------------------------------------------
#include <stdint.h>
//-----------------------------------------------------------
typedef uint64_t clock_t;
//-----------------------------------------------------------
#endif
```

95.1.5 lib/ctype.h

Si veda la sezione 91.3.

```
#ifndef _CTYPE_H
#define _CTYPE_H       1
//-----------------------------------------------------------
#include <NULL.h>
//-----------------------------------------------------------
#define isblank(C)  ((int) (C == ' ' || C == '\t'))
#define isspace(C)  ((int) (C == ' ' \
                            || C == '\f' \
                            || C == '\n' \
                            || C == '\n' \
                            || C == '\r' \
                            || C == '\t' \
                            || C == '\v'))
#define isdigit(C)  ((int) (C >= '0' && C <= '9'))
#define isxdigit(C) \
    ((int) ((C >= '0' && C <= '9') \
            || (C >= 'A' && C <= 'F') \
            || (C >= 'a' && C <= 'f')))
#define isupper(C)  ((int) (C >= 'A' && C <= 'Z'))
#define islower(C)  ((int) (C >= 'a' && C <= 'z'))
#define iscntrl(C)  ((int) ((C >= 0x00 && C <= 0x1F) \
                            || C == 0x7F))
#define isgraph(C)  ((int) (C >= 0x21 && C <= 0x7E))
#define isprint(C)  ((int) (C >= 0x20 && C <= 0x7E))
#define isalpha(C)  (isupper (C) || islower (C))
#define isalnum(C)  (isalpha (C) || isdigit (C))
#define ispunct(C)  (isgraph (C) && (!isspace (C)) \
                     && (!isalnum (C)))
#define tolower(C)  (isupper (C) ? ((C) + 0x20) : (C))
#define toupper(C)  (islower (C) ? ((C) - 0x20) : (C))
#define toascii(C)  (C & 0x7F)
#define _tolower(C) (isupper (C) ? ((C) + 0x20) : (C))
#define _toupper(C) (islower (C) ? ((C) - 0x20) : (C))
//-----------------------------------------------------------
#endif
```

95.1.6 lib/limits.h

Si veda la sezione 91.3.

```
#ifndef _LIMITS_H
#define _LIMITS_H       1
//-----------------------------------------------------------
#define CHAR_UNSIGNED   0
//-----------------------------------------------------------
#define CHAR_BIT        (8)
//
#define SCHAR_MIN       (-0x80)
#define SCHAR_MAX        (0x7F)
#define UCHAR_MAX        (0xFF)
//
#ifdef CHAR_UNSIGNED
#define CHAR_MIN      (0)
#define CHAR_MAX       UCHAR_MAX
#else
```



```
#define CHAR_MIN       SCHAR_MIN
#define CHAR_MAX       SCHAR_MAX
#endif
//
#define MB_LEN_MAX       (16)
//
#define SHRT_MIN        (-0x8000)
#define SHRT_MAX         (0x7FFF)
#define USHRT_MAX        (0xFFFF)
//
#define INT_MIN         (-0x80000000)
#define INT_MAX          (0x7FFFFFFF)
#define UINT_MAX         (0xFFFFFFFFU)
//
#define LONG_MIN        (-0x80000000L)
#define LONG_MAX         (0x7FFFFFFFL)
#define ULONG_MAX        (0xFFFFFFFFUL)
//
#define LLONG_MIN       (-0x8000000000000000LL)
#define LLONG_MAX        (0x7FFFFFFFFFFFFFFFLL)
#define ULLONG_MAX       (0xFFFFFFFFFFFFFFFFULL)
#define WORD_BIT         (32)
#define LONG_BIT         (32)
#define SSIZE_MAX        (0x7FFFFFFFL)
//-----------------------------------------------------------
#define ARG_MAX      8192       // Arguments+environment
                                // max length.
#define ATEXIT_MAX     32       // Max "at exit"
                                // functions.
#define FILESIZEBITS   32       // File size needs integer
                                // size...
#define LINK_MAX      254       // Max links per file.
#define NAME_MAX       14       // File name max
                                // (Minix 1 fs).
#define OPEN_MAX      128       // Max open files per
                                // process.
#define PATH_MAX     1024       // Path, including
                                // final '\0'.
#define MAX_CANON     256       // Max bytes in
                                // canonical tty queue.
#define MAX_INPUT       1       // Max bytes in tty
                                // input queue.
//-----------------------------------------------------------
#define CHLD_MAX          INT_MAX       // Not used.
#define HOST_NAME_MAX     INT_MAX       // Not used.
#define LOGIN_NAME_MAX    INT_MAX       // Not used.
#define PAGE_SIZE         INT_MAX       // Not used.
#define RE_DUP_MAX        INT_MAX       // Not used.
#define STREAM_MAX        INT_MAX       // Not used.
#define SYMLOOP_MAX       INT_MAX       // Not used.
#define TTY_NAME_MAX      INT_MAX       // Not used.
#define TZNAME_MAX        INT_MAX       // Not used.
#define PIPE_MAX          INT_MAX       // Not used.
#define SYMLINK_MAX       INT_MAX       // Not used.
//-----------------------------------------------------------
#endif
```

95.1.7 lib/ptrdiff_t.h

Si veda la sezione 91.3.

```
#ifndef _PTRDIFF_T_H
#define _PTRDIFF_T_H    1
//-----------------------------------------------------------
typedef int ptrdiff_t;
//-----------------------------------------------------------
#endif
```

95.1.8 lib/restrict.h

Si veda la sezione 91.3.

```
#ifndef _RESTRICT_H
#define _RESTRICT_H      1
//-----------------------------------------------------------
// At the moment, the GCC compiler does not support
// the 'restrict' keyword.
//-----------------------------------------------------------
#define restrict /**/
//-----------------------------------------------------------
#endif
```

95.1.9 lib/size_t.h

«

Si veda la sezione 91.3.

```
3230001 #ifndef _SIZE_T_H
3230002 #define _SIZE_T_H       1
3230003 //-------------------------------------------------------
3230004 // The type 'size_t' *must* be equal to an 'int'.
3230005 //-------------------------------------------------------
3230006 typedef unsigned int size_t;
3230007 //-------------------------------------------------------
3230008 #endif
```

95.1.10 lib/stdarg.h

«

Si veda la sezione 91.3.

```
3240001 #ifndef _STDARG_H
3240002 #define _STDARG_H       1
3240003 //-------------------------------------------------------
3240004 typedef unsigned char *va_list;
3240005 //-------------------------------------------------------
3240006 #define va_start(ap, last) \
3240007     ((void) ((ap) = \
3240008         ((va_list) &(last)) + (sizeof (last))))
3240009 #define va_end(ap) ((void) ((ap) = 0))
3240010 #define va_copy(dest, src) \
3240011     ((void) ((dest) = (va_list) (src)))
3240012 #define va_arg(ap, type) \
3240013     (((ap) = (ap) + (sizeof (type))), \
3240014         *((type *) ((ap) - (sizeof (type)))))
3240015 //-------------------------------------------------------
3240016 #endif
```

95.1.11 lib/stdbool.h

«

Si veda la sezione 91.3.

```
3250001 #ifndef _STDBOOL_H
3250002 #define _STDBOOL_H      1
3250003 //-------------------------------------------------------
3250004 #define bool    _Bool
3250005 #define true    1
3250006 #define false   0
3250007 #define __bool_true_false_are_defined   1
3250008 //-------------------------------------------------------
3250009 #endif
```

95.1.12 lib/stddef.h

«

Si veda la sezione 91.3.

```
3260001 #ifndef _STDDEF_H
3260002 #define _STDDEF_H      1
3260003 //-------------------------------------------------------
3260004 #include <ptrdiff_t.h>
3260005 #include <size_t.h>
3260006 #include <wchar_t.h>
3260007 #include <NULL.h>
3260008 //-------------------------------------------------------
3260009 #define offsetof(type, member) \
3260010     ((size_t) &((type *)0)->member)
3260011 //-------------------------------------------------------
3260012 #endif
```

95.1.13 lib/stdint.h

«

Si veda la sezione 91.3.

```
3270001 #ifndef _STDINT_H
3270002 #define _STDINT_H       1
3270003 //-------------------------------------------------------
3270004 typedef signed char int8_t;
3270005 typedef short int int16_t;
3270006 typedef int int32_t;
3270007 typedef long long int int64_t;
3270008 //
3270009 typedef unsigned char uint8_t;
3270010 typedef unsigned short int uint16_t;
3270011 typedef unsigned int uint32_t;
3270012 typedef unsigned long long int uint64_t;
3270013 //
3270014 #define INT8_MIN              (-0x80)
3270015 #define INT16_MIN             (-0x8000)
3270016 #define INT32_MIN             (-0x80000000)
3270017 #define INT64_MIN             (-0x8000000000000000LL)
3270018 //
3270019 #define INT8_MAX               0x7F
3270020 #define INT16_MAX              0x7FFF
3270021 #define INT32_MAX              0x7FFFFFFF
3270022 #define INT64_MAX              0x7FFFFFFFFFFFFFFFLL
3270023 //
3270024 #define UINT8_MAX              0xFF
3270025 #define UINT16_MAX             0xFFFF
3270026 #define UINT32_MAX             0xFFFFFFFFU
3270027 #define UINT64_MAX             0xFFFFFFFFFFFFFFFFULL
3270028 //-------------------------------------------------------
3270029 typedef signed char int_least8_t;
3270030 typedef short int int_least16_t;
3270031 typedef int int_least32_t;
3270032 typedef long long int int_least64_t;
3270033 //
3270034 typedef unsigned char uint_least8_t;
3270035 typedef unsigned short int uint_least16_t;
3270036 typedef unsigned int uint_least32_t;
3270037 typedef unsigned long long int uint_least64_t;
3270038 //
3270039 #define INT_LEAST8_MIN        (-0x80)
3270040 #define INT_LEAST16_MIN       (-0x8000)
3270041 #define INT_LEAST32_MIN       (-0x80000000)
3270042 #define INT_LEAST64_MIN       (-0x8000000000000000LL)
3270043 //
3270044 #define INT_LEAST8_MAX          0x7F
3270045 #define INT_LEAST16_MAX         0x7FFF
3270046 #define INT_LEAST32_MAX         0x7FFFFFFF
3270047 #define INT_LEAST64_MAX         0x7FFFFFFFFFFFFFFFLL
3270048 //
3270049 #define UINT_LEAST8_MAX         0xFF
3270050 #define UINT_LEAST16_MAX        0xFFFF
3270051 #define UINT_LEAST32_MAX        0xFFFFFFFFU
3270052 #define UINT_LEAST64_MAX        0xFFFFFFFFFFFFFFFFULL
3270053 //-------------------------------------------------------
3270054 #define INT8_C(VAL)             VAL
3270055 #define INT16_C(VAL)            VAL
3270056 #define INT32_C(VAL)            VAL
3270057 #define INT64_C(VAL)            VAL ## LL
3270058 //
3270059 #define UINT8_C(VAL)            VAL
3270060 #define UINT16_C(VAL)           VAL
3270061 #define UINT32_C(VAL)           VAL ## U
3270062 #define UINT64_C(VAL)           VAL ## ULL
3270063 //-------------------------------------------------------
3270064 typedef signed char int_fast8_t;
3270065 typedef int int_fast16_t;
3270066 typedef int int_fast32_t;
3270067 typedef long long int int_fast64_t;
3270068 //
3270069 typedef unsigned char uint_fast8_t;
3270070 typedef unsigned int uint_fast16_t;
3270071 typedef unsigned int uint_fast32_t;
3270072 typedef unsigned long long int uint_fast64_t;
3270073 //
3270074 #define INT_FAST8_MIN         (-0x80)
3270075 #define INT_FAST16_MIN        (-0x80000000)
3270076 #define INT_FAST32_MIN        (-0x80000000)
3270077 #define INT_FAST64_MIN        (-0x8000000000000000LL)
3270078 //
3270079 #define INT_FAST8_MAX           0x7F
3270080 #define INT_FAST16_MAX          0x7FFFFFFF
3270081 #define INT_FAST32_MAX          0x7FFFFFFF
3270082 #define INT_FAST64_MAX          0x7FFFFFFFFFFFFFFFLL
3270083 //
3270084 #define UINT_FAST8_MAX          0xFF
3270085 #define UINT_FAST16_MAX         0xFFFFFFFFU
3270086 #define UINT_FAST32_MAX         0xFFFFFFFFU
3270087 #define UINT_FAST64_MAX         0xFFFFFFFFFFFFFFFFULL
3270088 //-------------------------------------------------------
3270089 typedef int intptr_t;
3270090 typedef unsigned int uintptr_t;
3270091 //
3270092 #define INTPTR_MIN            (-0x80000000)
3270093 #define INTPTR_MAX              0x7FFFFFFF
3270094 #define UINTPTR_MAX             0xFFFFFFFFU
3270095 //
3270096 typedef long long int intmax_t;
3270097 typedef unsigned long long int uintmax_t;
3270098 //
3270099 #define INTMAX_C(VAL)           VAL ## LL
3270100 #define UINTMAX_C(VAL)          VAL ## ULL
3270101 #define INTMAX_MIN   (-INTMAX_C(0x8000000000000000))
3270102 #define INTMAX_MAX    (INTMAX_C(0x7FFFFFFFFFFFFFFF))
3270103 #define UINTMAX_MAX  (UINTMAX_C(0xFFFFFFFFFFFFFFFF))
3270104 //-------------------------------------------------------
3270105 #define PTRDIFF_MIN           (-0x80000000)
```

```
#define PTRDIFF_MAX             0x7FFFFFFF
//
#define SIG_ATOMIC_MIN        (-0x80000000)
#define SIG_ATOMIC_MAX          0x7FFFFFFF
//
#define SIZE_MAX                0xFFFFFFFFU
//
#define WCHAR_MIN               0x00000000
#define WCHAR_MAX               0xFFFFFFFFU
//
#define WINT_MIN              (-0x8000000000000000LL)
#define WINT_MAX                0x7FFFFFFFFFFFFFFFLL
//------------------------------------------------------------
#endif
```

95.1.14 lib/time_t.h

Si veda la sezione 91.3.

```
#ifndef _TIME_T_H
#define _TIME_T_H       1
//------------------------------------------------------------
typedef long long int time_t;
//------------------------------------------------------------
#endif
```

95.1.15 lib/wchar_t.h

Si veda la sezione 91.3.

```
#ifndef _WCHAR_T_H
#define _WCHAR_T_H      1
//------------------------------------------------------------
typedef unsigned int wchar_t;
//------------------------------------------------------------
#endif
```

95.2 os32: «lib/_gcc.h»

Si veda la sezione 88.1.

```
#ifndef __GCC_H
#define __GCC_H        1
//------------------------------------------------------------
#include <stdlib.h>
//------------------------------------------------------------
typedef struct
{
  unsigned long long int quot;
  unsigned long long int rem;
} ulldiv_t;
//------------------------------------------------------------
lldiv_t _lldiv (long long int dividend,
                long long int divisor);
ulldiv_t _ulldiv (unsigned long long int dividend,
                  unsigned long long int divisor);
//------------------------------------------------------------
unsigned long long int __udivdi3 (unsigned long long
                                  int dividend,
                                  unsigned long long
                                  int divisor);
unsigned long long int __umoddi3 (unsigned long long
                                  int dividend,
                                  unsigned long long
                                  int divisor);
long long int __divdi3 (long long int dividend,
                        long long int divisor);
long long int __moddi3 (long long int dividend,
                        long long int divisor);
//------------------------------------------------------------
#endif
```

95.2.1 lib/_gcc/__divdi3.c 759
95.2.2 lib/_gcc/__moddi3.c 759
95.2.3 lib/_gcc/__udivdi3.c 759
95.2.4 lib/_gcc/__umoddi3.c 759
95.2.5 lib/_gcc/_lldiv.c 759
95.2.6 lib/_gcc/_ulldiv.c 760

95.2.1 lib/_gcc/__divdi3.c

Si veda la sezione 88.1.

```
#include <_gcc.h>
//------------------------------------------------------------
long long int
__divdi3 (long long int dividend, long long int divisor)
{
  lldiv_t result;
  result = _lldiv (dividend, divisor);
  return result.quot;
}
```

95.2.2 lib/_gcc/__moddi3.c

Si veda la sezione 88.1.

```
#include <_gcc.h>
//------------------------------------------------------------
long long int
__moddi3 (long long int dividend, long long int divisor)
{
  lldiv_t result;
  result = _lldiv (dividend, divisor);
  return result.rem;
}
```

95.2.3 lib/_gcc/__udivdi3.c

Si veda la sezione 88.1.

```
#include <_gcc.h>
//------------------------------------------------------------
unsigned long long int
__udivdi3 (unsigned long long int dividend,
           unsigned long long int divisor)
{
  ulldiv_t result;
  result = _ulldiv (dividend, divisor);
  return result.quot;
}
```

95.2.4 lib/_gcc/__umoddi3.c

Si veda la sezione 88.1.

```
#include <_gcc.h>
//------------------------------------------------------------
unsigned long long int
__umoddi3 (unsigned long long int dividend,
           unsigned long long int divisor)
{
  ulldiv_t result;
  result = _ulldiv (dividend, divisor);
  return result.rem;
}
```

95.2.5 lib/_gcc/_lldiv.c

Si veda la sezione 88.1.

```
#include <_gcc.h>
//------------------------------------------------------------
// If DIVIDEND and DIVISOR have different sign,
// the QUOTIENT is negative.
//
// The REMINDER has the same sign as the DIVISOR.
//------------------------------------------------------------
lldiv_t
_lldiv (long long int dividend, long long int divisor)
{
  ulldiv_t uresult;
  lldiv_t result;
  //
  // Check for sign.
  //
  if (dividend >= 0 && divisor >= 0)
    {
      uresult = _ulldiv ((unsigned long long) dividend,
                         (unsigned long long) divisor);
      result.quot = uresult.quot;
      result.rem = uresult.rem;
    }
  else if (dividend < 0 && divisor < 0)
    {
      uresult =
        _ulldiv ((unsigned long long) -dividend,
```

```
3350027                   (unsigned long long) -divisor);
3350028       result.quot = uresult.quot;
3350029       result.rem = -uresult.rem;
3350030     }
3350031   else if (dividend < 0 && divisor >= 0)
3350032     {
3350033       uresult =
3350034         _ulldiv ((unsigned long long) -dividend,
3350035                  (unsigned long long) divisor);
3350036       result.quot = -uresult.quot;
3350037       result.rem = uresult.rem;
3350038     }
3350039   else if (dividend >= 0 && divisor < 0)
3350040     {
3350041       uresult = _ulldiv ((unsigned long long) dividend,
3350042                          (unsigned long long) -divisor);
3350043       result.quot = uresult.quot;
3350044       result.rem = -uresult.rem;
3350045     }
3350046   //
3350047   return (result);
3350048 }
```

95.2.6 lib/_gcc/_ulldiv.c

«

Si veda la sezione 88.1.

```
3360001 #include <_gcc.h>
3360002 //-------------------------------------------------------------
3360003 // DIVIDEND = DIVISOR * QUOTIENT + REMINDER
3360004 //
3360005 // If DIVISOR == 0,
3360006 // then QUOTIENT == 0 and REMINDER == DIVIDEND
3360007 //-------------------------------------------------------------
3360008 ulldiv_t
3360009 _ulldiv (unsigned long long int dividend,
3360010          unsigned long long int divisor)
3360011 {
3360012   unsigned long long int sign;
3360013   unsigned long long int mask;
3360014   ulldiv_t result;
3360015   int scroll;
3360016   unsigned int size;    // Bits of a long long.
3360017   //
3360018   // Division of zero will return zero.
3360019   //
3360020   if (dividend == 0)
3360021     {
3360022       result.quot = 0;
3360023       result.rem = 0;
3360024       return (result);
3360025     }
3360026   //
3360027   // Division by zero will return zero and all
3360028   // reminder.
3360029   //
3360030   if (divisor == 0)
3360031     {
3360032       result.quot = 0;
3360033       result.rem = dividend;
3360034       return (result);
3360035     }
3360036   //
3360037   // Calculate how much bits does have the type 'long
3360038   // long'.
3360039   //
3360040   size = 0;
3360041   mask = ~0LL;
3360042   //
3360043   while (mask > 0)
3360044     {
3360045       size += 8;
3360046       mask >>= 8;
3360047     }
3360048   //
3360049   // Calculate the value for 'sign' that needs to have
3360050   // the most
3360051   // significant bit to one.
3360052   //
3360053   mask = ~0LL;
3360054   mask >>= 1;
3360055   sign = ~mask;
3360056   //
3360057   // Scroll divisor to the left, as long as the first
3360058   // bit is zero.
3360059   //
3360060   for (scroll = 0; scroll < size; scroll++)
3360061     {
3360062       if (divisor & sign)
3360063         {
3360064           //
3360065           // The most significant bit is one.
3360066           //
3360067           break;
3360068         }
3360069       //
3360070       // The most significant bit is zero: scroll
3360071       // left.
3360072       //
3360073       divisor <<= 1;
3360074     }
3360075   //
3360076   //
3360077   //
3360078   result.quot = 0;
3360079   result.rem = 0;
3360080   //
3360081   for (; scroll >= 0 && divisor > 0; scroll--)
3360082     {
3360083       result.quot <<= 1;
3360084       if (dividend >= divisor)
3360085         {
3360086           result.quot |= 1LL;
3360087           dividend -= divisor;
3360088         }
3360089       divisor >>= 1;
3360090     }
3360091   //
3360092   result.rem = dividend;
3360093   //
3360094   return (result);
3360095 }
```

95.3 os32: «lib/arpa/inet.h»

«

Si veda la sezione 91.3.

```
3370001 #ifndef _ARPA_INET_H
3370002 #define _ARPA_INET_H       1
3370003 //-------------------------------------------------------------
3370004 #include <stdint.h>
3370005 #include <sys/socklen_t.h>
3370006 //-------------------------------------------------------------
3370007 uint32_t htonl (uint32_t host32);
3370008 uint16_t htons (uint16_t host16);
3370009 uint32_t ntohl (uint32_t net32);
3370010 uint16_t ntohs (uint16_t net16);
3370011 //-------------------------------------------------------------
3370012 const char *inet_ntop (int family, const void *src,
3370013                        char *dst, socklen_t size);
3370014 int inet_pton (int family, const char *src, void *dst);
3370015 //-------------------------------------------------------------
3370016 #endif
```


95.3.1 lib/arpa/inet/htonl.c 761
95.3.2 lib/arpa/inet/htons.c 762
95.3.3 lib/arpa/inet/inet_ntop.c 762
95.3.4 lib/arpa/inet/inet_pton.c 762
95.3.5 lib/arpa/inet/ntohl.c 764
95.3.6 lib/arpa/inet/ntohs.c 764

95.3.1 lib/arpa/inet/htonl.c

«

Si veda la sezione 88.11.

```
3380001 #include <arpa/inet.h>
3380002 //-------------------------------------------------------------
3380003 uint32_t
3380004 htonl (uint32_t host32)
3380005 {
3380006   uint8_t *orig = (void *) &host32;
3380007   union
3380008   {
3380009     uint32_t value;
3380010     uint8_t b[4];
3380011   } dest;
3380012   //
3380013   // Convert: must revert byte order.
3380014   //
3380015   dest.b[0] = orig[3];
```

```
3380016   dest.b[1] = orig[2];
3380017   dest.b[2] = orig[1];
3380018   dest.b[3] = orig[0];
3380019   //
3380020   return (dest.value);
3380021 }
```

95.3.2 lib/arpa/inet/htons.c

«

Si veda la sezione 88.11.

```
3390001 #include <arpa/inet.h>
3390002 //-------------------------------------------------------
3390003 uint16_t
3390004 htons (uint16_t host16)
3390005 {
3390006   uint8_t *orig = (void *) &host16;
3390007   union
3390008   {
3390009     uint16_t value;
3390010     uint8_t b[2];
3390011   } dest;
3390012   //
3390013   // Convert: must revert byte order.
3390014   //
3390015   dest.b[0] = orig[1];
3390016   dest.b[1] = orig[0];
3390017   //
3390018   return (dest.value);
3390019 }
```

95.3.3 lib/arpa/inet/inet_ntop.c

«

Si veda la sezione 88.66.

```
3400001 #include <arpa/inet.h>
3400002 #include <stdint.h>
3400003 #include <errno.h>
3400004 #include <string.h>
3400005 #include <stdlib.h>
3400006 //-------------------------------------------------------
3400007 const char *
3400008 inet_ntop (int family, const void *src, char *dst,
3400009            socklen_t size)
3400010 {
3400011   //
3400012   // Check family type: only IPv4 is available here.
3400013   //
3400014   if (family != AF_INET)
3400015     {
3400016       errset (EAFNOSUPPORT);
3400017       return (NULL);
3400018     }
3400019   //
3400020   // Check for NULL pointers.
3400021   //
3400022   if (src == NULL || dst == NULL)
3400023     {
3400024       errset (EINVAL);
3400025       return (NULL);
3400026     }
3400027   //
3400028   snprintf (dst, (size_t) size, "%i.%i.%i.%i",
3400029             *((in_addr_t *) src) >> 0 & 0x000000FF,
3400030             *((in_addr_t *) src) >> 8 & 0x000000FF,
3400031             *((in_addr_t *) src) >> 16 & 0x000000FF,
3400032             *((in_addr_t *) src) >> 24 & 0x000000FF);
3400033   //
3400034   // Return ok.
3400035   //
3400036   return (dst);
3400037 }
```

95.3.4 lib/arpa/inet/inet_pton.c

«

Si veda la sezione 88.67.

```
3410001 #include <arpa/inet.h>
3410002 #include <stdint.h>
3410003 #include <errno.h>
3410004 #include <string.h>
3410005 #include <stdlib.h>
3410006 //-------------------------------------------------------
3410007 #define INET_PTON_MAX_STRING_SIZE 31
3410008 //-------------------------------------------------------
3410009 int
3410010 inet_pton (int family, const char *src, void *dst)
```



```
3410011 {
3410012   char *t;
3410013   int ipv4[4];
3410014   int i;
3410015   in_addr_t result;
3410016   char source[INET_PTON_MAX_STRING_SIZE + 1];
3410017   //
3410018   // Check family type: only IPv4 is available here.
3410019   //
3410020   if (family != AF_INET)
3410021     {
3410022       errset (EAFNOSUPPORT);
3410023       return (-1);
3410024     }
3410025   //
3410026   // Check for NULL pointers.
3410027   //
3410028   if (src == NULL || dst == NULL)
3410029     {
3410030       errset (EINVAL);
3410031       return (-1);
3410032     }
3410033   //
3410034   // Check the source string size.
3410035   //
3410036   if (strlen (src) > INET_PTON_MAX_STRING_SIZE)
3410037     {
3410038       //
3410039       // The IPv4 address scan is finished
3410040       // prematurely:
3410041       // return zero to tell that the address string
3410042       // is
3410043       // not correct.
3410044       //
3410045       return (0);
3410046     }
3410047   //
3410048   // Copy the source address, to be able to modify
3410049   // the string.
3410050   //
3410051   strcpy (source, src);
3410052   //
3410053   // Start ''tokenize'' the string: it is here
3410054   // accepted also
3410055   // the space as a delimiter.
3410056   //
3410057   t = strtok (source, ". ");
3410058   //
3410059   for (i = 0; i < 4 && t != NULL; i++)
3410060     {
3410061       ipv4[i] = atoi (t);
3410062       if (ipv4[i] > 255 || ipv4[i] < 0)
3410063         {
3410064           //
3410065           // An octet cannot have a value greater than
3410066           // 255,
3410067           // and cannot be negative.
3410068           //
3410069           break;
3410070         }
3410071       t = strtok (NULL, ". ");
3410072     }
3410073   //
3410074   if (i < 4)
3410075     {
3410076       //
3410077       // The IPv4 address scan is finished
3410078       // prematurely:
3410079       // return zero to tell that the address string
3410080       // is
3410081       // not correct.
3410082       //
3410083       return (0);
3410084     }
3410085   //
3410086   // Translate into a network byte order IPv4 address:
3410087   // the architecture is little-endian.
3410088   //
3410089   result = 0;
3410090   result += (ipv4[0] << 0) & 0x000000FF;
3410091   result += (ipv4[1] << 8) & 0x0000FF00;
3410092   result += (ipv4[2] << 16) & 0x00FF0000;
3410093   result += (ipv4[3] << 24) & 0xFF000000;
3410094   //
3410095   // Update the destination.
3410096   //
3410097   *((in_addr_t *) dst) = result;
```

```
3410098   //
3410099   // Return ok.
3410100   //
3410101   return (1);
3410102 }
```

95.3.5 lib/arpa/inet/ntohl.c

Si veda la sezione 88.11.

```
3420001 #include <arpa/inet.h>
3420002 //-------------------------------------------------------------
3420003 uint32_t
3420004 ntohl (uint32_t net32)
3420005 {
3420006   uint8_t *orig = (void *) &net32;
3420007   union
3420008   {
3420009     uint32_t value;
3420010     uint8_t b[4];
3420011   } dest;
3420012   //
3420013   // Convert: must revert byte order.
3420014   //
3420015   dest.b[0] = orig[3];
3420016   dest.b[1] = orig[2];
3420017   dest.b[2] = orig[1];
3420018   dest.b[3] = orig[0];
3420019   //
3420020   return (dest.value);
3420021 }
```

95.3.6 lib/arpa/inet/ntohs.c

Si veda la sezione 88.11.

```
3430001 #include <arpa/inet.h>
3430002 //-------------------------------------------------------------
3430003 uint16_t
3430004 ntohs (uint16_t net16)
3430005 {
3430006   uint8_t *orig = (void *) &net16;
3430007   union
3430008   {
3430009     uint16_t value;
3430010     uint8_t b[2];
3430011   } dest;
3430012   //
3430013   // Convert: must revert byte order.
3430014   //
3430015   dest.b[0] = orig[1];
3430016   dest.b[1] = orig[0];
3430017   //
3430018   return (dest.value);
3430019 }
```

95.4 os32: «lib/dirent.h»

Si veda la sezione 91.3.

```
3440001 #ifndef _DIRENT_H
3440002 #define _DIRENT_H        1
3440003
3440004 #include <sys/types.h>  // ino_t
3440005 #include <limits.h>     // NAME_MAX
3440006
3440007 //-------------------------------------------------------------
3440008 struct dirent
3440009 {
3440010   ino_t d_ino;  // I-node number [1]
3440011   char d_name[NAME_MAX + 1];    // NAME_MAX + Null
3440012   // termination
3440013 } __attribute__ ((packed));
3440014 //
3440015 // [1] The type 'ino_t' must be equal to 'uint16_t',
3440016 //     because the directory inside the Minix 1 file
3440017 //     system has exactly such size.
3440018 //
3440019 //-------------------------------------------------------------
3440020 #define DOPEN_MAX   OPEN_MAX/2  // <limits.h> [1]
3440021 //
3440022 // [1] DOPEN_MAX is not standard, but it is used to
3440023 //     define how many directory slot to keep for open
3440024 //     directories. As directory streams are opened as
3440025 //     file descriptors, the sum of all kind of file
3440026 //     open cannot be more than OPEM_MAX.
3440027 //-------------------------------------------------------------
```



```
3440028 typedef struct
3440029 {
3440030   int fdn;      // File descriptor number.
3440031   struct dirent dir;    // Last directory item read.
3440032 } DIR;
3440033
3440034 extern DIR _directory_stream[]; // Defined inside
3440035                                 // 'lib/dirent/DIR.c'.
3440036 //-------------------------------------------------------------
3440037 // Function prototypes.
3440038 //-------------------------------------------------------------
3440039 int closedir (DIR * dp);
3440040 DIR *opendir (const char *name);
3440041 struct dirent *readdir (DIR * dp);
3440042 void rewinddir (DIR * dp);
3440043 //-------------------------------------------------------------
3440044
3440045 #endif
```

95.4.1 lib/dirent/DIR.c 765
95.4.2 lib/dirent/closedir.c 765
95.4.3 lib/dirent/opendir.c 766
95.4.4 lib/dirent/readdir.c 767
95.4.5 lib/dirent/rewinddir.c 768

95.4.1 lib/dirent/DIR.c

Si veda la sezione 91.3.

```
3450001 #include <dirent.h>
3450002 //
3450003 // There must be room for at least 'DOPEN_MAX'
3450004 // elements.
3450005 //
3450006 DIR _directory_stream[DOPEN_MAX];
3450007
3450008 void
3450009 _dirent_directory_stream_setup (void)
3450010 {
3450011   int d;
3450012   //
3450013   for (d = 0; d < DOPEN_MAX; d++)
3450014     {
3450015       _directory_stream[d].fdn = -1;
3450016     }
3450017 }
```

95.4.2 lib/dirent/closedir.c

Si veda la sezione 88.13.

```
3460001 #include <dirent.h>
3460002 #include <fcntl.h>
3460003 #include <sys/types.h>
3460004 #include <sys/stat.h>
3460005 #include <unistd.h>
3460006 #include <errno.h>
3460007 #include <stddef.h>
3460008 //-------------------------------------------------------------
3460009 int
3460010 closedir (DIR * dp)
3460011 {
3460012   //
3460013   // Check for a valid argument
3460014   //
3460015   if (dp == NULL)
3460016     {
3460017       //
3460018       // Not a valid pointer.
3460019       //
3460020       errset (EBADF);   // Invalid directory.
3460021       return (-1);
3460022     }
3460023   //
3460024   // Check if it is an open directory stream.
3460025   //
3460026   if (dp->fdn < 0)
3460027     {
3460028       //
3460029       // The stream is closed.
3460030       //
3460031       errset (EBADF);   // Invalid directory.
3460032       return (-1);
3460033     }
```

```
3460034   //
3460035   // Close the file descriptor. It there is an error,
3460036   // the 'errno' variable will be set by 'close()'.
3460037   //
3460038   return (close (dp->fdn));
3460039 }
```

95.4.3 lib/dirent/opendir.c

«

Si veda la sezione 88.89.

```
3470001 #include <dirent.h>
3470002 #include <fcntl.h>
3470003 #include <stdio.h>
3470004 #include <sys/types.h>
3470005 #include <sys/stat.h>
3470006 #include <unistd.h>
3470007 #include <errno.h>
3470008 #include <stddef.h>
3470009 //-------------------------------------------------------
3470010 DIR *
3470011 opendir (const char *path)
3470012 {
3470013   int fdn;
3470014   int d;
3470015   DIR *dp;
3470016   struct stat file_status;
3470017   //
3470018   // Function 'opendir()' is used only for reading.
3470019   //
3470020   fdn = open (path, O_RDONLY);
3470021   //
3470022   // Check the file descriptor returned.
3470023   //
3470024   if (fdn < 0)
3470025     {
3470026       //
3470027       // The variable 'errno' is already set:
3470028       // EINVAL
3470029       // EMFILE
3470030       // ENFILE
3470031       //
3470032       errset (errno);
3470033       return (NULL);
3470034     }
3470035   //
3470036   // Set the 'FD_CLOEXEC' flag for that file
3470037   // descriptor.
3470038   //
3470039   if (fcntl (fdn, F_SETFD, FD_CLOEXEC) != 0)
3470040     {
3470041       //
3470042       // The variable 'errno' is already set:
3470043       // EBADF
3470044       //
3470045       errset (errno);
3470046       close (fdn);
3470047       return (NULL);
3470048     }
3470049   //
3470050   //
3470051   //
3470052   if (fstat (fdn, &file_status) != 0)
3470053     {
3470054       //
3470055       // Error should be already set.
3470056       //
3470057       errset (errno);
3470058       close (fdn);
3470059       return (NULL);
3470060     }
3470061   //
3470062   // Verify it is a directory
3470063   //
3470064   if (!S_ISDIR (file_status.st_mode))
3470065     {
3470066       //
3470067       // It is not a directory!
3470068       //
3470069       close (fdn);
3470070       errset (ENOTDIR); // Is not a directory.
3470071       return (NULL);
3470072     }
3470073   //
3470074   // A valid file descriptor is available: must find a
3470075   // free
3470076   // '_directory_stream[]' slot.
```



```
3470077   //
3470078   for (d = 0; d < DOPEN_MAX; d++)
3470079     {
3470080       if (_directory_stream[d].fdn < 0)
3470081         {
3470082           //
3470083           // Found a free slot: set it up.
3470084           //
3470085           dp = &(_directory_stream[d]);
3470086           dp->fdn = fdn;
3470087           //
3470088           // Return the directory pointer.
3470089           //
3470090           return (dp);
3470091         }
3470092     }
3470093   //
3470094   // If we are here, there was no free directory slot
3470095   // available.
3470096   //
3470097   close (fdn);
3470098   errset (EMFILE);      // Too many file open.
3470099   return (NULL);
3470100 }
```

95.4.4 lib/dirent/readdir.c

«

Si veda la sezione 88.98.

```
3480001 #include <dirent.h>
3480002 #include <fcntl.h>
3480003 #include <sys/types.h>
3480004 #include <sys/stat.h>
3480005 #include <unistd.h>
3480006 #include <errno.h>
3480007 #include <stddef.h>
3480008 //-------------------------------------------------------
3480009 struct dirent *
3480010 readdir (DIR * dp)
3480011 {
3480012   ssize_t size;
3480013   //
3480014   // Check for a valid argument.
3480015   //
3480016   if (dp == NULL)
3480017     {
3480018       //
3480019       // Not a valid pointer.
3480020       //
3480021       errset (EBADF);   // Invalid directory.
3480022       return (NULL);
3480023     }
3480024   //
3480025   // Check if it is an open directory stream.
3480026   //
3480027   if (dp->fdn < 0)
3480028     {
3480029       //
3480030       // The stream is closed.
3480031       //
3480032       errset (EBADF);   // Invalid directory.
3480033       return (NULL);
3480034     }
3480035   //
3480036   // Read the directory.
3480037   //
3480038   size = read (dp->fdn, &(dp->dir), (size_t) 16);
3480039   //
3480040   // Fix the null termination, if the name is very
3480041   // long.
3480042   //
3480043   dp->dir.d_name[NAME_MAX] = '\0';
3480044   //
3480045   // Check what was read.
3480046   //
3480047   if (size == 0)
3480048     {
3480049       //
3480050       // End of directory, but it is not an error.
3480051       //
3480052       return (NULL);
3480053     }
3480054   //
3480055   if (size < 0)
3480056     {
3480057       //
3480058       // This is an error. The variable 'errno' is
```

```
      // already set.
      //
      errset (errno);
      return (NULL);
    }
  //
  if (dp->dir.d_ino == 0)
    {
      //
      // This is a null directory record.
      // Should try to read the next one.
      //
      return (readdir (dp));
    }
  //
  if (strlen (dp->dir.d_name) == 0)
    {
      //
      // This is a bad directory record: try to read
      // next.
      //
      return (readdir (dp));
    }
  //
  // A valid directory record should be available now.
  //
  return (&(dp->dir));
}
```

95.4.5 lib/dirent/rewinddir.c

«

Si veda la sezione 88.101.

```
#include <dirent.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include <stddef.h>
#include <stdio.h>
//-----------------------------------------------------------
void
rewinddir (DIR * dp)
{
  FILE *fp;
  //
  // Check for a valid argument.
  //
  if (dp == NULL)
    {
      //
      // Nothing to rewind, and no error to set.
      //
      return;
    }
  //
  // Check if it is an open directory stream.
  //
  if (dp->fdn < 0)
    {
      //
      // The stream is closed.
      // Nothing to rewind, and no error to set.
      //
      return;
    }
  //
  //
  //
  fp = &_stream[dp->fdn];
  //
  rewind (fp);
}
```

95.5 os32: «lib/errno.h»

«

Si veda la sezione 88.20.

```
#ifndef _ERRNO_H
#define _ERRNO_H       1
//-----------------------------------------------------------
#include <limits.h>
#include <string.h>
#include <sys/os32.h>
#include <kernel/lib_k.h>

```



```
//-----------------------------------------------------------
// The variable 'errno' is standard, but 'errln' and
// 'errfn' are added to keep track of the error source.
// Variable 'errln' is used to save the source file
// line number; variable 'errfn' is used to save the
// source file name. To set these variable in a
// consistent way it is also added a macroinstruction:
// 'errset'.
//-----------------------------------------------------------
extern int errno;
extern int errln;
extern char errfn[PATH_MAX];
//
#define errset(e) \
    (errln = __LINE__, \
     strncpy (errfn, __FILE__, PATH_MAX), \
     errno = e)
//-----------------------------------------------------------
// Standard POSIX 'errno' macro variables.
//-----------------------------------------------------------
#define E2BIG            1      // Argument list too
                                // long.
#define EACCES           2      // Permission denied.
#define EADDRINUSE       3      // Address in use.
#define EADDRNOTAVAIL    4      // Address not
                                // available.
#define EAFNOSUPPORT     5      // Address family not
                                // supported.
#define EAGAIN           6      // Resource
                                // unavailable, try
                                // again.
#define EALREADY         7      // Connection already
                                // in progress.
#define EBADF            8      // Bad file
                                // descriptor.
#define EBADMSG          9      // Bad message.
#define EBUSY           10      // Device or resource
                                // busy.
#define ECANCELED       11      // Operation canceled.
#define ECHILD          12      // No child processes.
#define ECONNABORTED    13      // Connection aborted.
#define ECONNREFUSED    14      // Connection refused.
#define ECONNRESET      15      // Connection reset.
#define EDEADLK         16      // Resource deadlock
                                // would occur.
#define EDESTADDRREQ    17      // Destination address
                                // required.
#define EDOM            18      // Mathematics
                                // argument out of
                                // domain of
                                // function.
#define EDQUOT          19      // Reserved.
#define EEXIST          20      // File exists.
#define EFAULT          21      // Bad address.
#define EFBIG           22      // File too large.
#define EHOSTUNREACH    23      // Host is
                                // unreachable.
#define EIDRM           24      // Identifier removed.
#define EILSEQ          25      // Illegal byte
                                // sequence.
#define EINPROGRESS     26      // Operation in
                                // progress.
#define EINTR           27      // Interrupted
                                // function.
#define EINVAL          28      // Invalid argument.
#define EIO             29      // I/O error.
#define EISCONN         30      // Socket is
                                // connected.
#define EISDIR          31      // Is a directory.
#define ELOOP           32      // Too many levels of
                                // symbolic links.
#define EMFILE          33      // Too many open
                                // files.
#define EMLINK          34      // Too many links.
#define EMSGSIZE        35      // Message too large.
#define EMULTIHOP       36      // Reserved.
#define ENAMETOOLONG    37      // Filename too long.
#define ENETDOWN        38      // Network is down.
#define ENETRESET       39      // Connection aborted
                                // by network.
#define ENETUNREACH     40      // Network
                                // unreachable.
#define ENFILE          41      // Too many files open
                                // in system.
#define ENOBUFS         42      // No buffer space
                                // available.
#define ENODATA         43      // No message is
```

```
3500096                                 // available on the
3500097                                 // stream head
3500098                                 // read queue.
3500099 #define ENODEV          44      // No such device.
3500100 #define ENOENT          45      // No such file or
3500101                                 // directory.
3500102 #define ENOEXEC         46      // Executable file
3500103                                 // format error.
3500104 #define ENOLCK          47      // No locks available.
3500105 #define ENOLINK         48      // Reserved.
3500106 #define ENOMEM          49      // Not enough space.
3500107 #define ENOMSG          50      // No message of the
3500108                                 // desired type.
3500109 #define ENOPROTOOPT     51      // Protocol not
3500110                                 // available.
3500111 #define ENOSPC          52      // No space left on
3500112                                 // device.
3500113 #define ENOSR           53      // No stream
3500114                                 // resources.
3500115 #define ENOSTR          54      // Not a stream.
3500116 #define ENOSYS          55      // Function not
3500117                                 // supported.
3500118 #define ENOTCONN        56      // The socket is not
3500119                                 // connected.
3500120 #define ENOTDIR         57      // Not a directory.
3500121 #define ENOTEMPTY       58      // Directory not
3500122                                 // empty.
3500123 #define ENOTSOCK        59      // Not a socket.
3500124 #define ENOTSUP         60      // Not supported.
3500125 #define ENOTTY          61      // Inappropriate I/O
3500126                                 // control operation.
3500127 #define ENXIO           62      // No such device or
3500128                                 // address.
3500129 #define EOPNOTSUPP      63      // Operation not
3500130                                 // supported on
3500131                                 // socket.
3500132 #define EOVERFLOW       64      // Value too large to
3500133                                 // be stored in data
3500134                                 // type.
3500135 #define EPERM           65      // Operation not
3500136                                 // permitted.
3500137 #define EPIPE           66      // Broken pipe.
3500138 #define EPROTO          67      // Protocol error.
3500139 #define EPROTONOSUPPORT 68      // Protocol not
3500140                                 // supported.
3500141 #define EPROTOTYPE      69      // Protocol wrong type
3500142                                 // for socket.
3500143 #define ERANGE          70      // Result too large.
3500144 #define EROFS           71      // Read-only file
3500145                                 // system.
3500146 #define ESPIPE          72      // Invalid seek.
3500147 #define ESRCH           73      // No such process.
3500148 #define ESTALE          74      // Reserved.
3500149 #define ETIME           75      // Stream ioctl()
3500150                                 // timeout.
3500151 #define ETIMEDOUT       76      // Connection timed
3500152                                 // out.
3500153 #define ETXTBSY         77      // Text file busy.
3500154 #define EWOULDBLOCK     78      // Operation would
3500155                                 // block (may be the
3500156                                 // same as EAGAIN).
3500157 #define EXDEV           79      // Cross-device link.
3500158 //------------------------------------------------------
3500159 // Added os32 errors.
3500160 //------------------------------------------------------
3500161 #define EUNKNOWN                (-1) // Unknown
3500162                                      // error.
3500163 #define E_NO_MEDIUM              80  // No medium
3500164                                      // found.
3500165 #define E_MEDIUM                 81  // Medium
3500166                                      // reported
3500167                                      // error.
3500168 #define E_FILE_TYPE              82  // File type
3500169                                      // not
3500170                                      // compatible.
3500171 #define E_ROOT_INODE_NOT_CACHED  83  // The root
3500172                                      // directory
3500173                                      // inode is
3500174                                      // not cached.
3500175 #define E_CANNOT_READ_SUPERBLOCK 84  // Cannot read
3500176                                      // super
3500177                                      // block.
3500178 #define E_MAP_INODE_TOO_BIG      85  // Map inode
3500179                                      // too big.
3500180 #define E_MAP_ZONE_TOO_BIG       86  // Map zone
3500181                                      // too big.
3500182 #define E_DATA_ZONE_TOO_BIG      87  // Data zone
```

```
3500183                                      // too big.
3500184 #define E_CANNOT_FIND_ROOT_DEVICE 88  // Cannot find
3500185                                      // root
3500186                                      // device.
3500187 #define E_CANNOT_FIND_ROOT_INODE  89  // Cannot find
3500188                                      // root inode.
3500189 #define E_FILE_TYPE_UNSUPPORTED   90  // File type
3500190                                      // unsupported.
3500191 #define E_ENV_TOO_BIG             91  // Environment
3500192                                      // too big.
3500193 #define E_LIMIT                   92  // Exceeded
3500194                                      // implementa-
3500195                                      // tion limits.
3500196 #define E_NOT_MOUNTED             93  // Not
3500197                                      // mounted.
3500198 #define E_NOT_IMPLEMENTED         94  // Not
3500199                                      // implemented.
3500200 #define E_HARDWARE_FAULT          95  // Hardware
3500201                                      // fault.
3500202 #define E_DRIVER_FAULT            96  // Driver
3500203                                      // fault.
3500204 #define E_PIPE_FULL               97  // Pipe full.
3500205 #define E_PIPE_EMPTY              98  // Pipe empty.
3500206 #define E_PART_TYPE_NOT_MINIX     99  // Not a Minix
3500207                                      // partition
3500208                                      // type.
3500209 #define E_FS_TYPE_NOT_SUPPORTED  100  // File system
3500210                                      // type not
3500211                                      // supported.
3500212 #define E_PDU_TOO_BIG            101  // PDU too
3500213                                      // big.
3500214 #define E_ARP_MISSING            102  // ARP missing
3500215                                      // address.
3500216 //------------------------------------------------------
3500217 // Default descriptions for errors.
3500218 //------------------------------------------------------
3500219 #define TEXT_E2BIG           "Argument list too long."
3500220 #define TEXT_EACCES          "Permission denied."
3500221 #define TEXT_EADDRINUSE      "Address in use."
3500222 #define TEXT_EADDRNOTAVAIL   "Address not available."
3500223 #define TEXT_EAFNOSUPPORT    "Address family not " \
3500224                              "supported."
3500225 #define TEXT_EAGAIN          "Resource unavailable, " \
3500226                              "try again."
3500227 #define TEXT_EALREADY        "Connection already in " \
3500228                              "progress."
3500229 #define TEXT_EBADF           "Bad file descriptor."
3500230 #define TEXT_EBADMSG         "Bad message."
3500231 #define TEXT_EBUSY           "Device or resource busy."
3500232 #define TEXT_ECANCELED       "Operation canceled."
3500233 #define TEXT_ECHILD          "No child processes."
3500234 #define TEXT_ECONNABORTED    "Connection aborted."
3500235 #define TEXT_ECONNREFUSED    "Connection refused."
3500236 #define TEXT_ECONNRESET      "Connection reset."
3500237 #define TEXT_EDEADLK         "Resource deadlock " \
3500238                              "would occur."
3500239 #define TEXT_EDESTADDRREQ    "Destination address " \
3500240                              "required."
3500241 #define TEXT_EDOM            "Mathematics argument " \
3500242                              "out of " \
3500243                              "domain of function."
3500244 #define TEXT_EDQUOT          "Reserved error: EDQUOT"
3500245 #define TEXT_EEXIST          "File exists."
3500246 #define TEXT_EFAULT          "Bad address."
3500247 #define TEXT_EFBIG           "File too large."
3500248 #define TEXT_EHOSTUNREACH    "Host is unreachable."
3500249 #define TEXT_EIDRM           "Identifier removed."
3500250 #define TEXT_EILSEQ          "Illegal byte sequence."
3500251 #define TEXT_EINPROGRESS     "Operation in progress."
3500252 #define TEXT_EINTR           "Interrupted function."
3500253 #define TEXT_EINVAL          "Invalid argument."
3500254 #define TEXT_EIO             "I/O error."
3500255 #define TEXT_EISCONN         "Socket is connected."
3500256 #define TEXT_EISDIR          "Is a directory."
3500257 #define TEXT_ELOOP           "Too many levels of " \
3500258                              "symbolic links."
3500259 #define TEXT_EMFILE          "Too many open files."
3500260 #define TEXT_EMLINK          "Too many links."
3500261 #define TEXT_EMSGSIZE        "Message too large."
3500262 #define TEXT_EMULTIHOP       "Reserved error: " \
3500263                              "EMULTIHOP"
3500264 #define TEXT_ENAMETOOLONG    "Filename too long."
3500265 #define TEXT_ENETDOWN        "Network is down."
3500266 #define TEXT_ENETRESET       "Connection aborted by " \
3500267                              "network."
3500268 #define TEXT_ENETUNREACH     "Network unreachable."
3500269 #define TEXT_ENFILE          "Too many files open " \
```

```
                                "in system."
#define TEXT_ENOBUFS            "No buffer space " \
                                "available."
#define TEXT_ENODATA            "No message is " \
                                "available on the " \
                                "stream head read queue."
#define TEXT_ENODEV             "No such device."
#define TEXT_ENOENT             "No such file or " \
                                "directory."
#define TEXT_ENOEXEC            "Executable file " \
                                "format error."
#define TEXT_ENOLCK             "No locks available."
#define TEXT_ENOLINK            "Reserved error: ENOLINK"
#define TEXT_ENOMEM             "Not enough space."
#define TEXT_ENOMSG             "No message of the " \
                                "desired type."
#define TEXT_ENOPROTOOPT        "Protocol not available."
#define TEXT_ENOSPC             "No space left on device."
#define TEXT_ENOSR              "No stream resources."
#define TEXT_ENOSTR             "Not a stream."
#define TEXT_ENOSYS             "Function not supported."
#define TEXT_ENOTCONN           "The socket is not " \
                                "connected."
#define TEXT_ENOTDIR            "Not a directory."
#define TEXT_ENOTEMPTY          "Directory not empty."
#define TEXT_ENOTSOCK           "Not a socket."
#define TEXT_ENOTSUP            "Not supported."
#define TEXT_ENOTTY             "Inappropriate I/O " \
                                "control operation."
#define TEXT_ENXIO              "No such device or " \
                                "address."
#define TEXT_EOPNOTSUPP         "Operation not " \
                                "supported on socket."
#define TEXT_EOVERFLOW          "Value too large to be " \
                                "stored in data type."
#define TEXT_EPERM              "Operation not permitted."
#define TEXT_EPIPE              "Broken pipe."
#define TEXT_EPROTO             "Protocol error."
#define TEXT_EPROTONOSUPPORT    "Protocol not supported."
#define TEXT_EPROTOTYPE         "Protocol wrong type " \
                                "for socket."
#define TEXT_ERANGE             "Result too large."
#define TEXT_EROFS              "Read-only file system."
#define TEXT_ESPIPE             "Invalid seek."
#define TEXT_ESRCH              "No such process."
#define TEXT_ESTALE             "Reserved error: ESTALE"
#define TEXT_ETIME              "Stream ioctl() timeout."
#define TEXT_ETIMEDOUT          "Connection timed out."
#define TEXT_ETXTBSY            "Text file busy."
#define TEXT_EWOULDBLOCK        "Operation would block."
#define TEXT_EXDEV              "Cross-device link."
//------------------------------------------------------------
#define TEXT_EUNKNOWN                   \
    "Unknown error."
#define TEXT_E_NO_MEDIUM                \
    "No medium found."
#define TEXT_E_MEDIUM                   \
    "Medium reported error"
#define TEXT_E_FILE_TYPE                \
    "File type not compatible."
#define TEXT_E_ROOT_INODE_NOT_CACHED    \
    "The root directory inode is not cached."
#define TEXT_E_CANNOT_READ_SUPERBLOCK   \
    "Cannot read super block."
#define TEXT_E_MAP_INODE_TOO_BIG        \
    "Map inode too big."
#define TEXT_E_MAP_ZONE_TOO_BIG         \
    "Map zone too big."
#define TEXT_E_DATA_ZONE_TOO_BIG        \
    "Data zone too big."
#define TEXT_E_CANNOT_FIND_ROOT_DEVICE  \
    "Cannot find root device."
#define TEXT_E_CANNOT_FIND_ROOT_INODE   \
    "Cannot find root inode."
#define TEXT_E_FILE_TYPE_UNSUPPORTED    \
    "File type unsupported."
#define TEXT_E_ENV_TOO_BIG              \
    "Environment too big."
#define TEXT_E_LIMIT                    \
    "Exceeded implementation limits."
#define TEXT_E_NOT_MOUNTED              \
    "Not mounted."
#define TEXT_E_NOT_IMPLEMENTED          \
    "Not implemented."
#define TEXT_E_HARDWARE_FAULT           \
    "Hardware fault."
#define TEXT_E_DRIVER_FAULT             \
```



```
    "Driver fault."
#define TEXT_E_PIPE_FULL                \
    "Pipe full."
#define TEXT_E_PIPE_EMPTY               \
    "Pipe empty."
#define TEXT_E_PART_TYPE_NOT_MINIX      \
    "Not a Minix partition type."
#define TEXT_E_FS_TYPE_NOT_SUPPORTED    \
    "File system type not supported."
#define TEXT_E_PDU_TOO_BIG              \
    "PDU too big."
#define TEXT_E_ARP_MISSING              \
    "ARP missing address."
//------------------------------------------------------------
#endif
```

95.5.1 lib/errno/errno.c 773

95.5.1 lib/errno/errno.c

Si veda la sezione 88.20.

```
//------------------------------------------------------------
// This file does not include the 'errno.h' header,
// because here 'errno' should not be declared as an
// extern variable!
//------------------------------------------------------------
#include <limits.h>
//------------------------------------------------------------
// The variable 'errno' is standard, but 'errln' and
// 'errfn' are added to keep track of the error source.
// Variable 'errln' is used to save the source file
// line number; variable 'errfn' is used to save the
// source file name.
// To set these variable in a consistent way it is
// also added a macroinstruction: 'errset'.
//------------------------------------------------------------
int errno;
int errln;
char errfn[PATH_MAX];
//------------------------------------------------------------
```

95.6 os32: «lib/fcntl.h»

Si veda la sezione 91.3.

```
#ifndef _FCNTL_H
#define _FCNTL_H        1

#include <sys/types.h>  // mode_t
                        // off_t
                        // pid_t
//------------------------------------------------------------
// Values for the second parameter of function
// 'fcntl()'.
//------------------------------------------------------------
#define F_DUPFD         0       // Duplicate file
                                // descriptor.
#define F_GETFD         1       // Get file descriptor
                                // flags.
#define F_SETFD         2       // Set file descriptor
                                // flags.
#define F_GETFL         3       // Get file status
                                // flags.
#define F_SETFL         4       // Set file status
                                // flags.
#define F_GETLK         5       // Get record locking
                                // information.
#define F_SETLK         6       // Set record locking
                                // information.
#define F_SETLKW        7       // Set record locking
                                // information;
                                // wait if blocked.
#define F_GETOWN        8       // Set owner of
                                // socket.
#define F_SETOWN        9       // Get owner of
                                // socket.
//------------------------------------------------------------
// Flags to be set with:
//   fcntl (fd, F_SETFD, ...);
//------------------------------------------------------------
#define FD_CLOEXEC      1       // Close the file
                                // descriptor upon
                                // execution of an
                                // exec() family
                                // function.
//------------------------------------------------------------
```

```
3520042 // Values for type 'l_type', used for record locking
3520043 // with 'fcntl()'.
3520044 //-------------------------------------------------------
3520045 #define F_RDLCK         0       // Read lock.
3520046 #define F_WRLCK         1       // Write lock.
3520047 #define F_UNLCK         2       // Remove lock.
3520048 //-------------------------------------------------------
3520049 // Flags for file creation, in place of 'oflag'
3520050 // parameter for function 'open()'.
3520051 //-------------------------------------------------------
3520052 #define O_CREAT         000010  // Create file if it
3520053                                 // does not exist.
3520054 #define O_EXCL          000020  // Exclusive use flag.
3520055 #define O_NOCTTY        000040  // Do not assign a
3520056                                 // controlling
3520057                                 // terminal.
3520058 #define O_TRUNC         000100  // Truncation flag.
3520059 //-------------------------------------------------------
3520060 // Flags for the file status, used with 'open()' and
3520061 // 'fcntl()'.
3520062 //-------------------------------------------------------
3520063 #define O_APPEND        000200  // Write append.
3520064 #define O_DSYNC         000400  // Synchronized write
3520065                                 // operations.
3520066 #define O_NONBLOCK      001000  // Non-blocking mode.
3520067 #define O_RSYNC         002000  // Synchronized read
3520068                                 // operations.
3520069 #define O_SYNC          004000  // Synchronized read
3520070                                 // and write.
3520071 //-------------------------------------------------------
3520072 // File access mask selection.
3520073 //-------------------------------------------------------
3520074 #define O_ACCMODE       000003  // Mask to select the
3520075                                 // last three bits,
3520076                                 // used to specify the
3520077                                 // main access
3520078                                 // modes: read, write
3520079                                 // and both.
3520080 //-------------------------------------------------------
3520081 // Main access modes.
3520082 //-------------------------------------------------------
3520083 #define O_RDONLY        000001  // Read.
3520084 #define O_WRONLY        000002  // Write.
3520085 #define O_RDWR         (O_RDONLY | O_WRONLY)    // [1]
3520086 //
3520087 // [1]  Both read and write.
3520088 //
3520089 //-------------------------------------------------------
3520090 // Structure 'flock', used to file lock for POSIX
3520091 // standard. It is not used inside os32.
3520092 //-------------------------------------------------------
3520093 struct flock
3520094 {
3520095   short int l_type;     // Type of lock: F_RDLCK,
3520096   // F_WRLCK, or F_UNLCK.
3520097   short int l_whence;   // Start reference point.
3520098   off_t l_start;        // Offset, from 'l_whence',
3520099   // for the area start.
3520100   off_t l_len;  // Locked area size. Zero means up to
3520101   // the end of the file.
3520102   pid_t l_pid;  // The process id blocking the area.
3520103 };
3520104 //-------------------------------------------------------
3520105 // Function prototypes.
3520106 //-------------------------------------------------------
3520107 int creat (const char *path, mode_t mode);
3520108 int fcntl (int fdn, int cmd, ...);
3520109 int open (const char *path, int oflags, ...);
3520110 //-------------------------------------------------------
3520111
3520112 #endif
```

95.6.1 lib/fcntl/creat.c 774
95.6.2 lib/fcntl/fcntl.c 775
95.6.3 lib/fcntl/open.c 775

95.6.1 lib/fcntl/creat.c

«

Si veda la sezione 88.14.

```
3530001 #include <fcntl.h>
3530002 #include <sys/types.h>
3530003 //-------------------------------------------------------
3530004 int
3530005 creat (const char *path, mode_t mode)
3530006 {
3530007   return (open (path, O_WRONLY | O_CREAT | O_TRUNC, mode));
3530008 }
```


95.6.2 lib/fcntl/fcntl.c

«

Si veda la sezione 87.18.

```
3540001 #include <fcntl.h>
3540002 #include <stdarg.h>
3540003 #include <stddef.h>
3540004 #include <string.h>
3540005 #include <errno.h>
3540006 #include <sys/os32.h>
3540007 #include <limits.h>
3540008 //-------------------------------------------------------
3540009 int
3540010 fcntl (int fdn, int cmd, ...)
3540011 {
3540012   va_list ap;
3540013   sysmsg_fcntl_t msg;
3540014   va_start (ap, cmd);
3540015   //
3540016   // Well known arguments.
3540017   //
3540018   msg.fdn = fdn;
3540019   msg.cmd = cmd;
3540020   //
3540021   // Select other arguments.
3540022   //
3540023   switch (cmd)
3540024     {
3540025     case F_DUPFD:
3540026     case F_SETFD:
3540027     case F_SETFL:
3540028       msg.arg = va_arg (ap, int);
3540029       break;
3540030     case F_GETFD:
3540031     case F_GETFL:
3540032       break;
3540033     case F_GETOWN:
3540034     case F_SETOWN:
3540035     case F_GETLK:
3540036     case F_SETLK:
3540037     case F_SETLKW:
3540038       errset (E_NOT_IMPLEMENTED);       // Not
3540039       // implemented.
3540040       return (-1);
3540041     default:
3540042       errset (EINVAL);  // Not implemented.
3540043       return (-1);
3540044     }
3540045   //
3540046   // Do the system call.
3540047   //
3540048   sys (SYS_FCNTL, &msg, (sizeof msg));
3540049   errno = msg.errno;
3540050   errln = msg.errln;
3540051   strncpy (errfn, msg.errfn, PATH_MAX);
3540052   return (msg.ret);
3540053 }
```

95.6.3 lib/fcntl/open.c

«

Si veda la sezione 87.37.

```
3550001 #include <fcntl.h>
3550002 #include <stdarg.h>
3550003 #include <stddef.h>
3550004 #include <string.h>
3550005 #include <errno.h>
3550006 #include <sys/os32.h>
3550007 #include <limits.h>
3550008 //-------------------------------------------------------
3550009 int
3550010 open (const char *path, int oflags, ...)
3550011 {
3550012   va_list ap;
3550013   sysmsg_open_t msg;
3550014   va_start (ap, oflags);
3550015   msg.path = path;
3550016   msg.flags = oflags;
3550017   msg.mode = va_arg (ap, mode_t);
3550018   sys (SYS_OPEN, &msg, (sizeof msg));
3550019   errno = msg.errno;
3550020   errln = msg.errln;
3550021   strncpy (errfn, msg.errfn, PATH_MAX);
3550022   return (msg.ret);
```

```
}
```

95.7 os32: «lib/grp.h»

Si veda la sezione 91.3.

```
#ifndef _GRP_H
#define _GRP_H        1
//-------------------------------------------------------
#include <restrict.h>
#include <sys/types.h>  // gid_t, uid_t
//-------------------------------------------------------
#define GR_MEM_MAX 32
struct group
{
  char *gr_name;
  char *gr_passwd;
  gid_t gr_gid;
  char *gr_mem[GR_MEM_MAX];
};
//-------------------------------------------------------
struct group *getgrent (void);
void setgrent (void);
void endgrent (void);
struct group *getgrnam (const char *name);
struct group *getgrgid (gid_t gid);
//-------------------------------------------------------
#endif
```

95.7.1 lib/grp/grent.c 776

95.7.1 lib/grp/grent.c

Si veda la sezione 88.53.

```
#include <grp.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
//-------------------------------------------------------
static char buffer[BUFSIZ];
static struct group gr;
static FILE *fp = NULL;
//-------------------------------------------------------
struct group *
getgrent (void)
{
  void *pstatus;
  char *char_gid;
  int i;
  //
  if (fp == NULL)
    {
      fp = fopen ("/etc/group", "r");
      if (fp == NULL)
        {
          return NULL;
        }
    }
  //
  pstatus = fgets (buffer, BUFSIZ, fp);
  if (pstatus == NULL)
    {
      return (NULL);
    }
  //
  // The parse is made with 'strtok()'. Please notice
  // that
  // 'strtok()' will not parse a line like the
  // following:
  // user::233:
  // The password field *must* have something,
  // otherwise the
  // GID will take the password place.
  // 'strtok()' will consider '::' the same as ':'!
  //
  gr.gr_name = strtok (buffer, ":");
  gr.gr_passwd = strtok (NULL, ":");
  char_gid = strtok (NULL, ":");
  for (i = 0; i < GR_MEM_MAX; i++)
    {
      gr.gr_mem[i] = strtok (NULL, ",\n");
    }
  gr.gr_gid = (gid_t) atoi (char_gid);
  //
  return (&gr);
}
```



```

//-------------------------------------------------------
void
endgrent (void)
{
  int status;
  //
  if (fp != NULL)
    {
      status = fclose (fp);
      if (status != 0)
        {
          perror (NULL);
          fp = NULL;
        }
    }
}

//-------------------------------------------------------
void
setgrent (void)
{
  if (fp != NULL)
    {
      rewind (fp);
    }
}

//-------------------------------------------------------
struct group *
getgrnam (const char *name)
{
  struct group *gr;
  //
  setgrent ();
  //
  for (;;)
    {
      gr = getgrent ();
      if (gr == NULL)
        {
          return (NULL);
        }
      if (strcmp (gr->gr_name, name) == 0)
        {
          return (gr);
        }
    }
}

//-------------------------------------------------------
struct group *
getgrgid (gid_t gid)
{
  struct group *gr;
  //
  setgrent ();
  //
  for (;;)
    {
      gr = getgrent ();
      if (gr == NULL)
        {
          return (NULL);
        }
      if (gr->gr_gid == gid)
        {
          return (gr);
        }
    }
}
```

95.8 os32: «lib/inttypes.h»

Si veda la sezione 91.3.

```
#ifndef _INTTYPES_H
#define _INTTYPES_H     1
//-------------------------------------------------------
#include <stdint.h>
#include <wchar_t.h>
#include <restrict.h>
//-------------------------------------------------------
typedef struct
{
  intmax_t quot;
  intmax_t rem;
```

```
3580012 } imaxdiv_t;
3580013 //
3580014 imaxdiv_t imaxdiv (intmax_t numer, intmax_t denom);
3580015 //-------------------------------------------------------
3580016 // Output typesetting.
3580017 //-------------------------------------------------------
3580018 #define PRId8           "d"
3580019 #define PRId16          "d"
3580020 #define PRId32          "d"
3580021 #define PRId64          "lld"
3580022 //
3580023 #define PRIdLEAST8      "d"
3580024 #define PRIdLEAST16     "d"
3580025 #define PRIdLEAST32     "d"
3580026 #define PRIdLEAST64     "lld"
3580027 //
3580028 #define PRIdFAST8       "d"
3580029 #define PRIdFAST16      "d"
3580030 #define PRIdFAST32      "d"
3580031 #define PRIdFAST64      "lld"
3580032 //
3580033 #define PRIdMAX         "lld"
3580034 #define PRIdPTR         "d"
3580035 //
3580036 #define PRIi8           "i"
3580037 #define PRIi16          "i"
3580038 #define PRIi32          "i"
3580039 #define PRIi64          "lli"
3580040 //
3580041 #define PRIiLEAST8      "i"
3580042 #define PRIiLEAST16     "i"
3580043 #define PRIiLEAST32     "i"
3580044 #define PRIiLEAST64     "lli"
3580045 //
3580046 #define PRIiFAST8       "i"
3580047 #define PRIiFAST16      "i"
3580048 #define PRIiFAST32      "i"
3580049 #define PRIiFAST64      "lli"
3580050 //
3580051 #define PRIiMAX         "lli"
3580052 #define PRIiPTR         "i"
3580053 //
3580054 #define PRIb8           "b"     // PRIb... is not
3580055                                 // standard!
3580056 #define PRIb16          "b"     //
3580057 #define PRIb32          "b"     //
3580058 #define PRIb64          "llb"   //
3580059 //                              //
3580060 #define PRIbLEAST8      "b"     //
3580061 #define PRIbLEAST16     "b"     //
3580062 #define PRIbLEAST32     "b"     //
3580063 #define PRIbLEAST64     "llb"   //
3580064 //                              //
3580065 #define PRIbFAST8       "b"     //
3580066 #define PRIbFAST16      "b"     //
3580067 #define PRIbFAST32      "b"     //
3580068 #define PRIbFAST64      "llb"   //
3580069 //                              //
3580070 #define PRIbMAX         "llb"   //
3580071 #define PRIbPTR         "b"     //
3580072 //
3580073 #define PRIo8           "o"
3580074 #define PRIo16          "o"
3580075 #define PRIo32          "o"
3580076 #define PRIo64          "llo"
3580077 //
3580078 #define PRIoLEAST8      "o"
3580079 #define PRIoLEAST16     "o"
3580080 #define PRIoLEAST32     "o"
3580081 #define PRIoLEAST64     "llo"
3580082 //
3580083 #define PRIoFAST8       "o"
3580084 #define PRIoFAST16      "o"
3580085 #define PRIoFAST32      "o"
3580086 #define PRIoFAST64      "llo"
3580087 //
3580088 #define PRIoMAX         "llo"
3580089 #define PRIoPTR         "o"
3580090 //
3580091 #define PRIu8           "u"
3580092 #define PRIu16          "u"
3580093 #define PRIu32          "u"
3580094 #define PRIu64          "llu"
3580095 //
3580096 #define PRIuLEAST8      "u"
3580097 #define PRIuLEAST16     "u"
3580098 #define PRIuLEAST32     "u"
```

```
3580099 #define PRIuLEAST64     "llu"
3580100 //
3580101 #define PRIuFAST8       "u"
3580102 #define PRIuFAST16      "u"
3580103 #define PRIuFAST32      "u"
3580104 #define PRIuFAST64      "llu"
3580105 //
3580106 #define PRIuMAX         "llu"
3580107 #define PRIuPTR         "u"
3580108 //
3580109 #define PRIx8           "x"
3580110 #define PRIx16          "x"
3580111 #define PRIx32          "x"
3580112 #define PRIx64          "llx"
3580113 //
3580114 #define PRIxLEAST8      "x"
3580115 #define PRIxLEAST16     "x"
3580116 #define PRIxLEAST32     "x"
3580117 #define PRIxLEAST64     "llx"
3580118 //
3580119 #define PRIxFAST8       "x"
3580120 #define PRIxFAST16      "x"
3580121 #define PRIxFAST32      "x"
3580122 #define PRIxFAST64      "llx"
3580123 //
3580124 #define PRIxMAX         "llx"
3580125 #define PRIxPTR         "x"
3580126 //
3580127 #define PRIX8           "X"
3580128 #define PRIX16          "X"
3580129 #define PRIX32          "X"
3580130 #define PRIX64          "llX"
3580131 //
3580132 #define PRIXLEAST8      "X"
3580133 #define PRIXLEAST16     "X"
3580134 #define PRIXLEAST32     "X"
3580135 #define PRIXLEAST64     "llX"
3580136 //
3580137 #define PRIXFAST8       "X"
3580138 #define PRIXFAST16      "X"
3580139 #define PRIXFAST32      "X"
3580140 #define PRIXFAST64      "llX"
3580141 //
3580142 #define PRIXMAX         "llX"
3580143 #define PRIXPTR         "X"
3580144 //-------------------------------------------------------
3580145 // Input scan and evaluation.
3580146 //-------------------------------------------------------
3580147 #define SCNd8           "hhd"
3580148 #define SCNd16          "hd"
3580149 #define SCNd32          "d"
3580150 #define SCNd64          "lld"
3580151 //
3580152 #define SCNdLEAST8      "hhd"
3580153 #define SCNdLEAST16     "hd"
3580154 #define SCNdLEAST32     "d"
3580155 #define SCNdLEAST64     "lld"
3580156 //
3580157 #define SCNdFAST8       "hhd"
3580158 #define SCNdFAST16      "d"
3580159 #define SCNdFAST32      "d"
3580160 #define SCNdFAST64      "lld"
3580161 //
3580162 #define SCNdMAX         "lld"
3580163 #define SCNdPTR         "d"
3580164 //
3580165 #define SCNi8           "hhi"
3580166 #define SCNi16          "hi"
3580167 #define SCNi32          "i"
3580168 #define SCNi64          "lli"
3580169 //
3580170 #define SCNiLEAST8      "hhi"
3580171 #define SCNiLEAST16     "hi"
3580172 #define SCNiLEAST32     "i"
3580173 #define SCNiLEAST64     "lli"
3580174 //
3580175 #define SCNiFAST8       "hhi"
3580176 #define SCNiFAST16      "i"
3580177 #define SCNiFAST32      "i"
3580178 #define SCNiFAST64      "lli"
3580179 //
3580180 #define SCNiMAX         "lli"
3580181 #define SCNiPTR         "i"
3580182 //
3580183 #define SCNb8           "hhb"   // SCNb... is not
3580184                                 // standard!
3580185 #define SCNb16          "hb"    //
```

```
3580186 #define SCNb32          "b"     //
3580187 #define SCNb64          "llb"   //
3580188 //                              //
3580189 #define SCNbLEAST8      "hhb"   //
3580190 #define SCNbLEAST16     "hb"    //
3580191 #define SCNbLEAST32     "b"     //
3580192 #define SCNbLEAST64     "llb"   //
3580193 //                              //
3580194 #define SCNbFAST8       "hhb"   //
3580195 #define SCNbFAST16      "b"     //
3580196 #define SCNbFAST32      "b"     //
3580197 #define SCNbFAST64      "llb"   //
3580198 //                              //
3580199 #define SCNbMAX         "llb"   //
3580200 #define SCNbPTR         "b"     //
3580201 //
3580202 #define SCNo8           "hho"
3580203 #define SCNo16          "ho"
3580204 #define SCNo32          "o"
3580205 #define SCNo64          "llo"
3580206 //
3580207 #define SCNoLEAST8      "hho"
3580208 #define SCNoLEAST16     "ho"
3580209 #define SCNoLEAST32     "o"
3580210 #define SCNoLEAST64     "llo"
3580211 //
3580212 #define SCNoFAST8       "hho"
3580213 #define SCNoFAST16      "o"
3580214 #define SCNoFAST32      "o"
3580215 #define SCNoFAST64      "llo"
3580216 //
3580217 #define SCNoMAX         "llo"
3580218 #define SCNoPTR         "o"
3580219 //
3580220 #define SCNu8           "hhu"
3580221 #define SCNu16          "hu"
3580222 #define SCNu32          "u"
3580223 #define SCNu64          "llu"
3580224 //
3580225 #define SCNuLEAST8      "hhu"
3580226 #define SCNuLEAST16     "hu"
3580227 #define SCNuLEAST32     "u"
3580228 #define SCNuLEAST64     "llu"
3580229 //
3580230 #define SCNuFAST8       "hhu"
3580231 #define SCNuFAST16      "u"
3580232 #define SCNuFAST32      "u"
3580233 #define SCNuFAST64      "llu"
3580234 //
3580235 #define SCNuMAX         "llu"
3580236 #define SCNuPTR         "u"
3580237 //
3580238 #define SCNx8           "hhx"
3580239 #define SCNx16          "hx"
3580240 #define SCNx32          "x"
3580241 #define SCNx64          "llx"
3580242 //
3580243 #define SCNxLEAST8      "hhx"
3580244 #define SCNxLEAST16     "hx"
3580245 #define SCNxLEAST32     "x"
3580246 #define SCNxLEAST64     "llx"
3580247 //
3580248 #define SCNxFAST8       "hhx"
3580249 #define SCNxFAST16      "x"
3580250 #define SCNxFAST32      "x"
3580251 #define SCNxFAST64      "llx"
3580252 //
3580253 #define SCNxMAX         "llx"
3580254 #define SCNxPTR         "x"
3580255 //---------------------------------------------------------
3580256 intmax_t imaxabs (intmax_t j);
3580257 intmax_t strtoimax (const char *restrict nptr,
3580258                     char **restrict endptr, int base);
3580259 uintmax_t strtouimax (const char *restrict nptr,
3580260                      char **restrict endptr, int base);
3580261 intmax_t wcstoimax (const wchar_t * restrict nptr,
3580262                     wchar_t ** restrict endptr, int base);
3580263 uintmax_t wcstouimax (const wchar_t * restrict nptr,
3580264                      wchar_t ** restrict endptr, int base);
3580265 //---------------------------------------------------------
3580266 #endif
```

95.8.1 lib/inttypes/imaxabs.c 781
95.8.2 lib/inttypes/imaxdiv.c 781

95.8.1 lib/inttypes/imaxabs.c

Si veda la sezione 88.3.

```
3590001 #include <inttypes.h>
3590002 //---------------------------------------------------------
3590003 intmax_t
3590004 imaxabs (intmax_t j)
3590005 {
3590006   if (j < 0)
3590007     {
3590008       return -j;
3590009     }
3590010   else
3590011     {
3590012       return j;
3590013     }
3590014 }
```

95.8.2 lib/inttypes/imaxdiv.c

Si veda la sezione 88.17.

```
3600001 #include <inttypes.h>
3600002 //---------------------------------------------------------
3600003 imaxdiv_t
3600004 imaxdiv (intmax_t numer, intmax_t denom)
3600005 {
3600006   imaxdiv_t d;
3600007   d.quot = numer / denom;
3600008   d.rem = numer % denom;
3600009   return d;
3600010 }
```

95.9 os32: «lib/libgen.h»

Si veda la sezione 91.3.

```
3610001 #ifndef _LIBGEN_H
3610002 #define _LIBGEN_H       1
3610003
3610004 //---------------------------------------------------------
3610005 char *basename (char *path);
3610006 char *dirname (char *path);
3610007 //---------------------------------------------------------
3610008
3610009 #endif
```

95.9.1 lib/libgen/basename.c 781
95.9.2 lib/libgen/dirname.c 782

95.9.1 lib/libgen/basename.c

Si veda la sezione 88.10.

```
3620001 #include <libgen.h>
3620002 #include <limits.h>
3620003 #include <stddef.h>
3620004 #include <string.h>
3620005 //---------------------------------------------------------
3620006 char *
3620007 basename (char *path)
3620008 {
3620009   static char *point = ".";     // When 'path' is
3620010   // NULL.
3620011   char *p;      // Pointer inside 'path'.
3620012   int i;        // Scan index inside 'path'.
3620013   //
3620014   // Empty path.
3620015   //
3620016   if (path == NULL || strlen (path) == 0)
3620017     {
3620018       return (point);
3620019     }
3620020   //
3620021   // Remove all final '/' if it exists, excluded the
3620022   // first character:
3620023   // 'i' is kept greater than zero.
3620024   //
3620025   for (i = (strlen (path) - 1);
3620026        i > 0 && path[i] == '/'; i--)
3620027     {
3620028       path[i] = 0;
3620029     }
3620030   //
3620031   // After removal of extra final '/', if there is
3620032   // only one '/', this
```

```
3620033   // is to be returned.
3620034   //
3620035   if (strncmp (path, "/", PATH_MAX) == 0)
3620036     {
3620037       return (path);
3620038     }
3620039   //
3620040   // If there are no '/'.
3620041   //
3620042   if (strchr (path, '/') == NULL)
3620043     {
3620044       return (path);
3620045     }
3620046   //
3620047   // Find the last '/' and calculate a pointer to the
3620048   // base name.
3620049   //
3620050   p = strrchr (path, (unsigned int) '/');
3620051   p++;
3620052   //
3620053   // Return the pointer to the base name.
3620054   //
3620055   return (p);
3620056 }
```

95.9.2 lib/libgen/dirname.c

«

Si veda la sezione 88.10.

```
3630001 #include <libgen.h>
3630002 #include <limits.h>
3630003 #include <stddef.h>
3630004 #include <string.h>
3630005 //-------------------------------------------------------------
3630006 char *
3630007 dirname (char *path)
3630008 {
3630009   static char *point = ".";     // When 'path' is
3630010   // NULL.
3630011   char *p;      // Pointer inside 'path'.
3630012   int i;        // Scan index inside 'path'.
3630013   //
3630014   // Empty path.
3630015   //
3630016   if (path == NULL || strlen (path) == 0)
3630017     {
3630018       return (point);
3630019     }
3630020   //
3630021   // Simple cases.
3630022   //
3630023   if (strncmp (path, "/", PATH_MAX) == 0 ||
3630024       strncmp (path, ".", PATH_MAX) == 0 ||
3630025       strncmp (path, "..", PATH_MAX) == 0)
3630026     {
3630027       return (path);
3630028     }
3630029   //
3630030   // Remove all final '/' if it exists, excluded the
3630031   // first character:
3630032   // 'i' is kept greater than zero.
3630033   //
3630034   for (i = (strlen (path) - 1);
3630035        i > 0 && path[i] == '/'; i--)
3630036     {
3630037       path[i] = 0;
3630038     }
3630039   //
3630040   // After removal of extra final '/', if there is
3630041   // only one '/', this
3630042   // is to be returned.
3630043   //
3630044   if (strncmp (path, "/", PATH_MAX) == 0)
3630045     {
3630046       return (path);
3630047     }
3630048   //
3630049   // If there are no '/'
3630050   //
3630051   if (strchr (path, '/') == NULL)
3630052     {
3630053       return (point);
3630054     }
3630055   //
3630056   // If there is only a '/' a the beginning.
3630057   //
3630058   if (path[0] == '/' &&
```



```
3630059       strchr (&path[1], (unsigned int) '/') == NULL)
3630060     {
3630061       path[1] = 0;
3630062       return (path);
3630063     }
3630064   //
3630065   // Replace the last '/' with zero.
3630066   //
3630067   p = strrchr (path, (unsigned int) '/');
3630068   *p = 0;
3630069   //
3630070   // Now remove extra duplicated final '/', except the
3630071   // very first
3630072   // character: 'i' is kept greater than zero.
3630073   //
3630074   for (i = (strlen (path) - 1);
3630075        i > 0 && path[i] == '/'; i--)
3630076     {
3630077       path[i] = 0;
3630078     }
3630079   //
3630080   // Now 'path' appears as a reduced string: the
3630081   // original path string
3630082   // is modified.
3630083   //
3630084   return (path);
3630085 }
```

95.10 os32: «lib/netinet/icmp.h»

«

Si veda la sezione 91.3.

```
3640001 #ifndef __NETINET_ICMP_H
3640002 #define __NETINET_ICMP_H    1
3640003 //-------------------------------------------------------------
3640004 // GNU C compatible ICMPv4 header and definitions
3640005 //-------------------------------------------------------------
3640006 #include <sys/types.h>
3640007 #include <netinet/in.h>
3640008 #include <netinet/ip.h>
3640009 //-------------------------------------------------------------
3640010 struct icmphdr
3640011 {
3640012   uint8_t type; // message type [1]
3640013   uint8_t code; // type sub-code [2]
3640014   uint16_t checksum;
3640015   union
3640016   {
3640017     struct
3640018     {
3640019       uint16_t id;
3640020       uint16_t sequence;
3640021     } __attribute__ ((packed)) echo;    // echo
3640022     // datagram
3640023     uint32_t gateway;   // gateway address
3640024     struct
3640025     {
3640026       uint16_t unused;
3640027       uint16_t mtu;
3640028     } __attribute__ ((packed)) frag;    // path mtu
3640029     // discovery
3640030   } un;
3640031 } __attribute__ ((packed));
3640032 //
3640033 // [1] message type:
3640034 //
3640035 #define ICMP_ECHOREPLY          0  // echo reply
3640036 #define ICMP_DEST_UNREACH       3  // destination
3640037                                    // unreachable
3640038 #define ICMP_SOURCE_QUENCH      4  // source
3640039                                    // quench
3640040 #define ICMP_REDIRECT           5  // redirect
3640041                                    // (change
3640042                                    // route)
3640043 #define ICMP_ECHO               8  // echo
3640044                                    // request
3640045 #define ICMP_TIME_EXCEEDED      11 // time
3640046                                    // exceeded
3640047 #define ICMP_PARAMETERPROB      12 // parameter
3640048                                    // problem
3640049 #define ICMP_TIMESTAMP          13 // timestamp
3640050                                    // request
3640051 #define ICMP_TIMESTAMPREPLY     14 // timestamp
3640052                                    // reply
3640053 #define ICMP_INFO_REQUEST       15 // information
3640054                                    // request
3640055 #define ICMP_INFO_REPLY         16 // information
```

```
3640056                                 // reply
3640057 #define ICMP_ADDRESS            17 // address
3640058                                    // mask
3640059                                    // request
3640060 #define ICMP_ADDRESSREPLY       18 // address
3640061                                    // mask reply
3640062 #define NR_ICMP_TYPES           18
3640063 //
3640064 // [2] type ICMP_DEST_UNREACH, code:
3640065 //
3640066 #define ICMP_NET_UNREACH        0  // network
3640067                                    // unreachable
3640068 #define ICMP_HOST_UNREACH       1  // host
3640069                                    // unreachable
3640070 #define ICMP_PROT_UNREACH       2  // protocol
3640071                                    // unreachable
3640072 #define ICMP_PORT_UNREACH       3  // port
3640073                                    // unreachable
3640074 #define ICMP_FRAG_NEEDED        4  // fragmentation
3640075                                    // needed/DF
3640076                                    // set
3640077 #define ICMP_SR_FAILED          5  // source
3640078                                    // route
3640079                                    // failed
3640080 #define ICMP_NET_UNKNOWN        6  // destination
3640081                                    // network
3640082                                    // unknown
3640083 #define ICMP_HOST_UNKNOWN       7  // destination
3640084                                    // host
3640085                                    // unknown
3640086 #define ICMP_HOST_ISOLATED      8  // source host
3640087                                    // isolated
3640088 #define ICMP_NET_ANO            9  // destination
3640089                                    // network
3640090                                    // administratively
3640091                                    // prohibited
3640092 #define ICMP_HOST_ANO           10 // destination
3640093                                    // host
3640094                                    // administratively
3640095                                    // prohibited
3640096 #define ICMP_NET_UNR_TOS        11 // network
3640097                                    // unreachable
3640098                                    // for this
3640099                                    // type of
3640100                                    // service
3640101 #define ICMP_HOST_UNR_TOS       12 // host
3640102                                    // unreachable
3640103                                    // for this
3640104                                    // type of
3640105                                    // service
3640106 #define ICMP_PKT_FILTERED       13 // packet
3640107                                    // filtered
3640108 #define ICMP_PREC_VIOLATION     14 // precedence
3640109                                    // violation
3640110 #define ICMP_PREC_CUTOFF        15 // precedence
3640111                                    // cut off
3640112 #define NR_ICMP_UNREACH         15 // instead of
3640113                                    // hardcoding
3640114                                    // immediate
3640115                                    // value
3640116 //
3640117 // [2] type ICMP_REDIRECT, code:
3640118 //
3640119 #define ICMP_REDIR_NET          0  // redirect
3640120                                    // net
3640121 #define ICMP_REDIR_HOST         1  // redirect
3640122                                    // host
3640123 #define ICMP_REDIR_NETTOS       2  // redirect
3640124                                    // net for TOS
3640125 #define ICMP_REDIR_HOSTTOS      3  // redirect
3640126                                    // host for
3640127                                    // TOS
3640128 //
3640129 // [2] type ICMP_TIME_EXCEEDED, code:
3640130 //
3640131 #define ICMP_EXC_TTL            0   // TTL count
3640132                                     // exceeded
3640133 #define ICMP_EXC_FRAGTIME       1   // fragment
3640134                                     // reass time
3640135                                     // exceeded
3640136 //------------------------------------------------------
3640137 #endif
```


95.11 os32: «lib/netinet/in.h»

«

Si veda la sezione 91.3.

```
3650001 #ifndef _NETINET_IN_H
3650002 #define _NETINET_IN_H    1
3650003 //------------------------------------------------------
3650004 #include <stdint.h>
3650005 #include <sys/sa_family_t.h>
3650006 //------------------------------------------------------
3650007 typedef uint16_t in_port_t;     // Port number. [1]
3650008 typedef uint32_t in_addr_t;     // IPv4 address.
3650009 //
3650010 // [1] Types 'in_port_t' and 'in_addr_t' are to be
3650011 //     intended for network byte order IPv4 integer
3650012 //     address, at least because this type is
3650013 //     used inside the type 'struct in_addr', that is
3650014 //     surely in network byte order. But attention must
3650015 //     be made to mistakes: for example,
3650016 //     inside the file <netinet/in.h> from GNU sources,
3650017 //     there are some macro defining default netmask
3650018 //     like this:
3650019 //
3650020 // #define IN_CLASSA(a)
3650021 //    ((((in_addr_t)(a)) & 0x80000000) == 0)
3650022 // #define IN_CLASSB(a)
3650023 //    ((((in_addr_t)(a)) & 0xc0000000) == 0x80000000)
3650024 // #define IN_CLASSC(a)
3650025 //    ((((in_addr_t)(a)) & 0xe0000000) == 0xc0000000)
3650026 //
3650027 //     Such macro can work only if the architecture is
3650028 //     big-endian.
3650029 //
3650030 //------------------------------------------------------
3650031 //
3650032 // IPv4 address.
3650033 //
3650034 struct in_addr
3650035 {
3650036   in_addr_t s_addr;
3650037 };
3650038 //
3650039 // struct sockaddr_in, members in *network*byte*order*.
3650040 //
3650041 struct sockaddr_in
3650042 {
3650043   sa_family_t sin_family;       // AF_INET.
3650044   in_port_t sin_port;   // Port number.
3650045   struct in_addr sin_addr;      // IP address.
3650046   uint8_t sin_zero[8];  // [2]
3650047 };
3650048 //
3650049 // [2] The type 'struct sockaddr_in' must be
3650050 //     replaceable with the type 'struct sockaddr',
3650051 //     with a cast. So it is necessary to fill the
3650052 //     unused space with a filler.
3650053 //
3650054 //------------------------------------------------------
3650055 //
3650056 // IPv6 address, network byte order.
3650057 //
3650058 struct in6_addr
3650059 {
3650060   uint8_t s6_addr[16];
3650061 };
3650062 //
3650063 // struct sockaddr_in6, members in network byte order.
3650064 //
3650065 struct sockaddr_in6
3650066 {
3650067   sa_family_t sin6_family;      // AF_INET6.
3650068   in_port_t sin6_port;  // Port number.
3650069   uint32_t sin6_flowinfo;       // IPv6 traffic class
3650070   // and flow info.
3650071   struct in6_addr sin6_addr;    // IPv6 address.
3650072   uint32_t sin6_scope_id;       // Set of interfaces
3650073   // for a scope.
3650074 };
3650075 //------------------------------------------------------
3650076 //external in6_addr in6addr_any;
3650077 //#define IN6ADDR_ANY_INIT  ...
3650078 //external struct in6_addr in6addr_loopback;
3650079 //#define IN6ADDR_LOOPBACK_INIT ...
3650080 //------------------------------------------------------
3650081 //
3650082 //
3650083 //
3650084 struct ipv6_mreq
```

```
3650085 {
3650086   struct in6_addr ipv6mr_multiaddr;     // IPv6
3650087   // multicast
3650088   // address.
3650089   unsigned int ipv6mr_interface;        // Interface
3650090   // index.
3650091 };
3650092 //-------------------------------------------------------
3650093 #define IPPROTO_IP      0       // Internet protocol.
3650094 #define IPPROTO_ICMP    1       // Contro message
3650095                                 // protocol.
3650096 #define IPPROTO_TCP     6       // Transmission
3650097                                 // control protocol.
3650098 #define IPPROTO_UDP    17       // User datagram
3650099                                 // protocol.
3650100 #define IPPROTO_IPV6   41       // Internet protocol
3650101                                 // version 6.
3650102 #define IPPROTO_RAW   255       // Raw IP packets
3650103                                 // protocol
3650104 //-------------------------------------------------------
3650105 //
3650106 // 0.0.0.0
3650107 //
3650108 #define INADDR_ANY         ((in_addr_t) 0x00000000)
3650109 //
3650110 // 255.255.255.255
3650111 //
3650112 #define INADDR_BROADCAST   ((in_addr_t) 0xffffffff)
3650113 //
3650114 // 127.0.0.1
3650115 //
3650116 #define INADDR_LOOPBACK    ((in_addr_t) 0x7f000001)
3650117 //
3650118 //
3650119 //
3650120 #define INET_ADDRSTRLEN  16     // IPv4 address string
3650121                                 // size.
3650122 #define INET6_ADDRSTRLEN 46     // IPv6 address string
3650123                                 // size.
3650124 //-------------------------------------------------------
3650125 #endif
```

95.12 os32: «lib/netinet/ip.h»

«

Si veda la sezione 91.3.

```
3660001 #ifndef _NETINET_IP_H
3660002 #define _NETINET_IP_H    1
3660003 //-------------------------------------------------------
3660004 // GNU C compatible IPv4 header.
3660005 //-------------------------------------------------------
3660006 #include <netinet/in.h>
3660007 //-------------------------------------------------------
3660008 struct iphdr
3660009 {
3660010   uint16_t ihl:4,       // header length / 4
3660011     version:4;  // IP version
3660012   uint8_t tos;  // type of service
3660013   uint16_t tot_len;     // total packet length
3660014   uint16_t id;  // identification
3660015   uint16_t frag_off;    // fragment offset field
3660016   uint8_t ttl;  // time to live
3660017   uint8_t protocol;     // contained protocol
3660018   uint16_t check;       // header checksum
3660019   in_addr_t saddr;      // source IP address
3660020   in_addr_t daddr;      // destination IP address
3660021   //
3660022   // Options after this point.
3660023   //
3660024 };
3660025 //-------------------------------------------------------
3660026 #define IPVERSION       4       // IP version number
3660027 #define IP_MAXPACKET    65535   // maximum packet size
3660028 //
3660029 #define MAXTTL          255     // maximum time to
3660030                                 // live (seconds)
3660031 #define IPDEFTTL        64      // default ttl, from
3660032                                 // RFC 1340
3660033 #define IPFRAGTTL       60      // time to live for
3660034                                 // fragments
3660035 #define IPTTLDEC        1       // subtracted when
3660036                                 // forwarding
3660037 //
3660038 #define IP_MSS          576     // default maximum
3660039                                 // segment size
```



```
3660040 //-------------------------------------------------------
3660041 #endif
```

95.13 os32: «lib/netinet/tcp.h»

«

Si veda la sezione 91.3.

```
3670001 #ifndef _NETINET_TCP_H
3670002 #define _NETINET_TCP_H  1
3670003 //-------------------------------------------------------
3670004 // GNU C compatible UDP header.
3670005 //-------------------------------------------------------
3670006 #include <sys/types.h>
3670007 //-------------------------------------------------------
3670008 struct tcphdr
3670009 {
3670010   uint16_t source;
3670011   uint16_t dest;
3670012   uint32_t seq;
3670013   uint32_t ack_seq;
3670014   uint16_t res1:4,
3670015     doff:4,
3670016     fin:1, syn:1, rst:1, psh:1, ack:1, urg:1, res2:2;
3670017   uint16_t window;
3670018   uint16_t check;
3670019   uint16_t urg_ptr;
3670020 };
3670021 //-------------------------------------------------------
3670022 // ATTENZIONE: per dare un significato allo stato di
3670023 // una connessione, occorre distinguere in che modo si
3670024 // trova inizialmente il socket:
3670025 // attivo o passivo (passivo quando rimane in ascolto
3670026 // per una connessione).
3670027 //
3670028 enum
3670029 {
3670030   TCP_LISTEN = 1,       // waiting a connection
3670031   // request
3670032   TCP_SYN_SENT, // SYN was sent, waiting from the
3670033   // response SYN
3670034   TCP_SYN_RECV, // SYN received, waiting for ACK
3670035   TCP_ESTABLISHED,      // SYN sent, SYN received and
3670036   // ACK sent
3670037   TCP_FIN_WAIT1,        // local close, FIN sent,
3670038   // waiting ACK or FIN
3670039   TCP_FIN_WAIT2,        // FIN sent, ACK received,
3670040   // waiting FIN
3670041   TCP_CLOSE_WAIT,       // FIN received, ACK sent,
3670042   // waiting local close
3670043   TCP_CLOSING,  // FIN sent, FIN received, ACK sent,
3670044   // waiting ACK
3670045   TCP_LAST_ACK, // FIN received, ACK and FIN sent,
3670046   // waiting ACK
3670047   TCP_TIME_WAIT,        // after TCP_LAST_ACK, wait a
3670048   // little and remove
3670049   TCP_CLOSE,    // connection removed
3670050   TCP_RESET     // connection reset (not standard)
3670051 };
3670052
3670053 #define TCPOPT_EOL              0
3670054 #define TCPOPT_NOP              1
3670055 #define TCPOPT_MAXSEG           2
3670056 #define TCPOLEN_MAXSEG          4
3670057 #define TCPOPT_WINDOW           3
3670058 #define TCPOLEN_WINDOW          3
3670059 #define TCPOPT_SACK_PERMITTED   4
3670060 #define TCPOLEN_SACK_PERMITTED  2
3670061 #define TCPOPT_SACK             5
3670062 #define TCPOPT_TIMESTAMP        8
3670063 #define TCPOLEN_TIMESTAMP       10
3670064 //-------------------------------------------------------
3670065 //
3670066 // TCP max segment size: IP_MSS - IP header size.
3670067 // Suppose to have a max IP header of 56 bytes,
3670068 // TCP_MSS == 520.
3670069 //
3670070 #define TCP_MSS         520
3670071 //-------------------------------------------------------
3670072 // LA STRUTTURA SEGUENTE È DA VALUTARE, forse conviene
3670073 // fare una tabella a parte per le connessioni TCP.
3670074 //
3670075 struct tcp_info
3670076 {
3670077   uint8_t tcpi_state;
3670078   uint8_t tcpi_ca_state;
```

```
3670079   uint8_t tcpi_retransmits;
3670080   uint8_t tcpi_probes;
3670081   uint8_t tcpi_backoff;
3670082   uint8_t tcpi_options;
3670083   uint8_t tcpi_snd_wscale:4, tcpi_rcv_wscale:4;
3670084
3670085   uint32_t tcpi_rto;
3670086   uint32_t tcpi_ato;
3670087   uint32_t tcpi_snd_mss;
3670088   uint32_t tcpi_rcv_mss;
3670089
3670090   uint32_t tcpi_unacked;
3670091   uint32_t tcpi_sacked;
3670092   uint32_t tcpi_lost;
3670093   uint32_t tcpi_retrans;
3670094   uint32_t tcpi_fackets;
3670095
3670096   /* Times. */
3670097   uint32_t tcpi_last_data_sent;
3670098
3670099   /* Not remembered, sorry.  */
3670100   uint32_t tcpi_last_ack_sent;
3670101
3670102   uint32_t tcpi_last_data_recv;
3670103   uint32_t tcpi_last_ack_recv;
3670104
3670105   /* Metrics. */
3670106   uint32_t tcpi_pmtu;
3670107   uint32_t tcpi_rcv_ssthresh;
3670108   uint32_t tcpi_rtt;
3670109   uint32_t tcpi_rttvar;
3670110   uint32_t tcpi_snd_ssthresh;
3670111   uint32_t tcpi_snd_cwnd;
3670112   uint32_t tcpi_advmss;
3670113   uint32_t tcpi_reordering;
3670114
3670115   uint32_t tcpi_rcv_rtt;
3670116   uint32_t tcpi_rcv_space;
3670117
3670118   uint32_t tcpi_total_retrans;
3670119 };
3670120
3670121
3670122 //-------------------------------------------------------
3670123 #endif
```

95.14 os32: «lib/netinet/udp.h»

Si veda la sezione 91.3.

```
3680001 #ifndef __NETINET_UDP_H
3680002 #define __NETINET_UDP_H    1
3680003 //-------------------------------------------------------
3680004 // GNU C compatible UDP header.
3680005 //-------------------------------------------------------
3680006 #include <sys/types.h>
3680007 //-------------------------------------------------------
3680008 struct udphdr
3680009 {
3680010   uint16_t source;      // source port
3680011   uint16_t dest;        // destination port
3680012   uint16_t len; // length
3680013   uint16_t check;       // checksum
3680014 } __attribute__ ((packed));
3680015 //-------------------------------------------------------
3680016 #endif
```

95.15 os32: «lib/pwd.h»

Si veda la sezione 91.3.

```
3690001 #ifndef _PWD_H
3690002 #define _PWD_H        1
3690003 //-------------------------------------------------------
3690004 #include <restrict.h>
3690005 #include <sys/types.h>  // gid_t, uid_t
3690006 //-------------------------------------------------------
3690007 struct passwd
3690008 {
3690009   char *pw_name;
3690010   char *pw_passwd;
3690011   uid_t pw_uid;
3690012   gid_t pw_gid;
3690013   char *pw_gecos;
3690014   char *pw_dir;
3690015   char *pw_shell;
3690016 };
3690017 //-------------------------------------------------------
3690018 struct passwd *getpwent (void);
3690019 void setpwent (void);
3690020 void endpwent (void);
3690021 struct passwd *getpwnam (const char *name);
3690022 struct passwd *getpwuid (uid_t uid);
3690023 //-------------------------------------------------------
3690024
3690025 #endif
```

95.15.1 lib/pwd/pwent.c 789

95.15.1 lib/pwd/pwent.c

Si veda la sezione 88.57.

```
3700001 #include <pwd.h>
3700002 #include <stdio.h>
3700003 #include <string.h>
3700004 #include <stdlib.h>
3700005 //-------------------------------------------------------
3700006 static char buffer[BUFSIZ];
3700007 static struct passwd pw;
3700008 static FILE *fp = NULL;
3700009 //-------------------------------------------------------
3700010 struct passwd *
3700011 getpwent (void)
3700012 {
3700013   void *pstatus;
3700014   char *char_uid;
3700015   char *char_gid;
3700016   //
3700017   if (fp == NULL)
3700018     {
3700019       fp = fopen ("/etc/passwd", "r");
3700020       if (fp == NULL)
3700021         {
3700022           return NULL;
3700023         }
3700024     }
3700025   //
3700026   pstatus = fgets (buffer, BUFSIZ, fp);
3700027   if (pstatus == NULL)
3700028     {
3700029       return (NULL);
3700030     }
3700031   //
3700032   // The parse is made with 'strtok()'. Please notice
3700033   // that
3700034   // 'strtok()' will not parse a line like the
3700035   // following:
3700036   // user::1001:233:...
3700037   // The password field *must* have something,
3700038   // otherwise the
3700039   // UID will take the password place.
3700040   // 'strtok()' will consider '::' the same as ':'!
3700041   //
3700042   pw.pw_name = strtok (buffer, ":");
3700043   pw.pw_passwd = strtok (NULL, ":");
3700044   char_uid = strtok (NULL, ":");
3700045   char_gid = strtok (NULL, ":");
3700046   pw.pw_gecos = strtok (NULL, ":");
3700047   pw.pw_dir = strtok (NULL, ":");
3700048   pw.pw_shell = strtok (NULL, "\n");
3700049   pw.pw_uid = (uid_t) atoi (char_uid);
3700050   pw.pw_gid = (gid_t) atoi (char_gid);
3700051   //
3700052   return (&pw);
3700053 }
3700054
3700055 //-------------------------------------------------------
3700056 void
3700057 endpwent (void)
3700058 {
3700059   int status;
3700060   //
3700061   if (fp != NULL)
3700062     {
3700063       status = fclose (fp);
3700064       if (status != 0)
3700065         {
3700066           perror (NULL);
```

```
3700067           fp = NULL;
3700068         }
3700069       else
3700070         {
3700071           ;     // printf ("[%s] fclose (fp)\n",
3700072           // __func__);
3700073         }
3700074     }
3700075 }
3700076
3700077 //-------------------------------------------------------
3700078 void
3700079 setpwent (void)
3700080 {
3700081   if (fp != NULL)
3700082     {
3700083       rewind (fp);
3700084     }
3700085 }
3700086
3700087 //-------------------------------------------------------
3700088 struct passwd *
3700089 getpwnam (const char *name)
3700090 {
3700091   struct passwd *pw;
3700092   //
3700093   setpwent ();
3700094   //
3700095   for (;;)
3700096     {
3700097       pw = getpwent ();
3700098       if (pw == NULL)
3700099         {
3700100           return (NULL);
3700101         }
3700102       if (strcmp (pw->pw_name, name) == 0)
3700103         {
3700104           return (pw);
3700105         }
3700106     }
3700107 }
3700108
3700109 //-------------------------------------------------------
3700110 struct passwd *
3700111 getpwuid (uid_t uid)
3700112 {
3700113   struct passwd *pw;
3700114   //
3700115   setpwent ();
3700116   //
3700117   for (;;)
3700118     {
3700119       pw = getpwent ();
3700120       if (pw == NULL)
3700121         {
3700122           return (NULL);
3700123         }
3700124       if (pw->pw_uid == uid)
3700125         {
3700126           return (pw);
3700127         }
3700128     }
3700129 }
```

95.16 os32: «lib/setjmp.h»

«

Si veda la sezione 87.49.

```
3710001 #ifndef _SETJMP_H
3710002 #define _SETJMP_H       1
3710003 //-------------------------------------------------------
3710004 #include <sys/os32.h>
3710005 #include <NULL.h>
3710006 //-------------------------------------------------------
3710007 typedef struct
3710008 {
3710009   uint32_t eax0;
3710010   uint32_t ecx0;
3710011   uint32_t edx0;
3710012   uint32_t ebx0;
3710013   uint32_t ebp0;
3710014   uint32_t esi0;
3710015   uint32_t edi0;
3710016   uint32_t ds0;
3710017   uint32_t es0;
3710018   uint32_t fs0;
3710019   uint32_t gs0;
3710020   uint32_t eip0;
3710021   uint32_t cs0;
3710022   uint32_t eflags0;
3710023   //
3710024   uint32_t eip1;
3710025   uint32_t syscallnr;
3710026   uint32_t msg_pointer;
3710027   uint32_t msg_size;
3710028   //
3710029   uint32_t env;
3710030   uint32_t ret;
3710031   uint32_t ebp1;
3710032   uint32_t eip2;
3710033   //
3710034 } jmp_stack_t;
3710035
3710036 typedef struct
3710037 {
3710038   uint32_t esp0;
3710039   uint32_t eax0;
3710040   uint32_t ecx0;
3710041   uint32_t edx0;
3710042   uint32_t ebx0;
3710043   uint32_t ebp0;
3710044   uint32_t esi0;
3710045   uint32_t edi0;
3710046   uint32_t ds0;
3710047   uint32_t es0;
3710048   uint32_t fs0;
3710049   uint32_t gs0;
3710050   uint32_t eip0;
3710051   uint32_t cs0;
3710052   uint32_t eflags0;
3710053   //
3710054   uint32_t eip1;
3710055   uint32_t syscallnr;
3710056   uint32_t msg_pointer;
3710057   uint32_t msg_size;
3710058   //
3710059   uint32_t env;
3710060   uint32_t ret;
3710061   uint32_t ebp1;
3710062   uint32_t eip2;
3710063   //
3710064 } jmp_env_t;
3710065 //
3710066 typedef char jmp_buf[sizeof (jmp_env_t)];
3710067 //-------------------------------------------------------
3710068 int setjmp (jmp_buf env);
3710069 void longjmp (jmp_buf env, int val);
3710070 //-------------------------------------------------------
3710071 #endif
```


95.16.1 lib/setjmp/longjmp.c 791
95.16.2 lib/setjmp/setjmp.s 791

95.16.1 lib/setjmp/longjmp.c

«

Si veda la sezione 87.49.

```
3720001 #include <sys/os32.h>
3720002 #include <setjmp.h>
3720003 //-------------------------------------------------------
3720004 void
3720005 longjmp (jmp_buf env, int val)
3720006 {
3720007   sysmsg_jmp_t msg;
3720008   msg.env = env;
3720009   msg.ret = val;
3720010   sys (SYS_LONGJMP, &msg, sizeof msg);
3720011 }
```

95.16.2 lib/setjmp/setjmp.s

«

Si veda la sezione 87.49.

```
3730001 .global setjmp
3730002 .extern sys
3730003 #-------------------------------------------------------
3730004 .text
3730005 #-------------------------------------------------------
3730006 .align 4
3730007 setjmp:
3730008     #
3730009     # Previous pushes:
3730010     #
```

```
3730011     #   push &env
3730012     #   push back_address   # made by a call to
3730013     #                       # setjmp() function
3730014     #
3730015     enter $8, $0
3730016     #
3730017     # sysmsg_jmp_t msg;
3730018     #
3730019     movl  $0,  -4(%ebp)        # msg.ret = 0;
3730020     #
3730021     movl  8(%ebp), %eax        # msg.env = env;
3730022     movl  %eax,-8(%ebp)
3730023     #
3730024     # sys (SYS_SETJMP, &msg, sizeof msg);
3730025     #
3730026     lea   -8(%ebp), %eax
3730027     pushl $8                    # sizeof msg
3730028     pushl %eax                  # &msg
3730029     pushl $47                   # SYS_SETJMP
3730030     call  sys
3730031     add   $4, %esp
3730032     add   $4, %esp
3730033     add   $4, %esp
3730034     #
3730035     # return (msg.ret);
3730036     #
3730037     movl  -4(%ebp), %eax
3730038     leave
3730039     ret
```

95.17 os32: «lib/signal.h»

Si veda la sezione 91.3.

```
3740001 #ifndef _SIGNAL_H
3740002 #define _SIGNAL_H       1
3740003 //-------------------------------------------------------
3740004 #include <sys/types.h>
3740005 //-------------------------------------------------------
3740006 #define SIGHUP           1
3740007 #define SIGINT           2
3740008 #define SIGQUIT          3
3740009 #define SIGILL           4
3740010 #define SIGABRT          6
3740011 #define SIGFPE           8
3740012 #define SIGKILL          9
3740013 #define SIGSEGV         11
3740014 #define SIGPIPE         13
3740015 #define SIGALRM         14
3740016 #define SIGTERM         15
3740017 #define SIGSTOP         17
3740018 #define SIGTSTP         18
3740019 #define SIGCONT         19
3740020 #define SIGCHLD         20
3740021 #define SIGTTIN         21
3740022 #define SIGTTOU         22
3740023 #define SIGUSR1         30
3740024 #define SIGUSR2         31
3740025 //-------------------------------------------------------
3740026 typedef int sig_atomic_t;
3740027 typedef void (*sighandler_t) (int);     // [1]
3740028 //
3740029 // [1] The type 'sighandler_t' is a pointer to a
3740030 // function for the signal handling, with a parameter
3740031 // of type 'int', returning 'void'.
3740032 //
3740033 //-------------------------------------------------------
3740034 // Special function used to call the real signal
3740035 // handler. This function will return to the 'back'
3740036 // address, instead where it was called.
3740037 //
3740038 void _sighandler_wrapper (uint32_t handler,
3740039                           uint32_t signal, uint32_t back);
3740040 //-------------------------------------------------------
3740041 // Special undeclarable functions.
3740042 //
3740043 #define SIG_ERR ((sighandler_t) -1)     // [2]
3740044 #define SIG_DFL ((sighandler_t) 0)      // [2]
3740045 #define SIG_IGN ((sighandler_t) 1)      // [2]
3740046 //
3740047 // [2] It transforms an integer number into a
3740048 //     'sighnandler_t' type, that is, a pointer
3740049 //     to a function that does not exists really.
3740050 //
3740051 //-------------------------------------------------------
3740052 sighandler_t signal (int sig, sighandler_t handler);
3740053 int kill (pid_t pid, int sig);
```



```
3740054 int raise (int sig);
3740055 //-------------------------------------------------------
3740056 #endif
```

95.17.1 lib/signal/_sighandler_wrapper.s 793
95.17.2 lib/signal/kill.c 794
95.17.3 lib/signal/signal.c 794

95.17.1 lib/signal/_sighandler_wrapper.s

Si veda la sezione 87.52.

```
3750001 .global _sighandler_wrapper
3750002 #-------------------------------------------------------
3750003 .section .text
3750004 #-------------------------------------------------------
3750005 # Port input byte.
3750006 #-------------------------------------------------------
3750007 _sighandler_wrapper:
3750008     #
3750009     # Current stack is:
3750010     #
3750011     # push %eip           # Back from interrupted code.
3750012     # push <sig_num>      # Signal number.
3750013     # push <sig_handler>  # Signal handler address
3750014     #
3750015     # Please note that THERE IS NO RETURN ADDRESS!
3750016     # Instead you find the signal handler address
3750017     # there.
3750018     #
3750019     # This routine should have to call the signal
3750020     # handler function, and then return back to the
3750021     # interrupted code.
3750022     #
3750023     enter $0, $0                # No local variables.
3750024     pushf
3750025     pusha
3750026     .equ SIG_HAND,  4           # First argument.  [1]
3750027     .equ SIG_NUM,   8           # Second argument. [1]
3750028     #
3750029     # [1] This function is called without the return
3750030     #     address inside the stack. So the arguments
3750031     #     are 4 bytes nearer than the usual.
3750032     #
3750033     mov  SIG_NUM(%ebp), %edx    # Copy the signal
3750034                                 # number into EDX.
3750035     mov  SIG_HAND(%ebp), %eax   # Copy the signal
3750036                                 # handler function
3750037                                 # address into EAX.
3750038     push %edx                   # Prepare argument for
3750039                                 # the signal
3750040                                 # handler function.
3750041     call *%eax                  # Call the signal
3750042                                 # handler function.
3750043     add  $4, %esp               # Pop the signal
3750044                                 # number argument.
3750045     popa
3750046     popf
3750047     leave
3750048     #
3750049     # Now we are back to the same stack as the
3750050     # beginning:
3750051     #
3750052     # push %eip         # back from interrupted code.
3750053     # push <sig_num>
3750054     # push <sig_handler>
3750055     # push %eip         # back from
3750056     #                   # _sighandler_wrapper()
3750057     #
3750058     # The stack pointer must be modified before
3750059     # returning, so that the address to the original
3750060     # interrupted instruction is used for return.
3750061     # Without such modification, the RET
3750062     # instruction would find the signal handler address
3750063     # instead!
3750064     #
3750065     add $4, %esp
3750066     add $4, %esp
3750067     #
3750068     # Now we are ready to return to the original
3750069     # interrupted address!
3750070     #
3750071     ret
3750072
```

95.17.2 lib/signal/kill.c

«

Si veda la sezione 87.29.

```
3760001 #include <sys/os32.h>
3760002 #include <sys/types.h>
3760003 #include <signal.h>
3760004 #include <errno.h>
3760005 #include <string.h>
3760006 //-----------------------------------------------------------
3760007 int
3760008 kill (pid_t pid, int sig)
3760009 {
3760010   sysmsg_kill_t msg;
3760011   if (pid < -1) // Currently unsupported.
3760012     {
3760013       errset (ESRCH);
3760014       return (-1);
3760015     }
3760016   msg.pid = pid;
3760017   msg.signal = sig;
3760018   msg.ret = 0;
3760019   msg.errno = 0;
3760020   sys (SYS_KILL, &msg, (sizeof msg));
3760021   errno = msg.errno;
3760022   errln = msg.errln;
3760023   strncpy (errfn, msg.errfn, PATH_MAX);
3760024   return (msg.ret);
3760025 }
```

95.17.3 lib/signal/signal.c

«

Si veda la sezione 87.52.

```
3770001 #include <sys/os32.h>
3770002 #include <sys/types.h>
3770003 #include <signal.h>
3770004 #include <errno.h>
3770005 #include <string.h>
3770006 //-----------------------------------------------------------
3770007 sighandler_t
3770008 signal (int sig, sighandler_t handler)
3770009 {
3770010   sysmsg_signal_t msg;
3770011
3770012   msg.signal = sig;
3770013   msg.handler = handler;
3770014   msg.wrapper = (uintptr_t) _sighandler_wrapper;
3770015   msg.ret = SIG_DFL;
3770016   msg.errno = 0;
3770017   sys (SYS_SIGNAL, &msg, (sizeof msg));
3770018   errno = msg.errno;
3770019   errln = msg.errln;
3770020   strncpy (errfn, msg.errfn, PATH_MAX);
3770021   return (msg.ret);
3770022 }
```

95.18 os32: «lib/stdio.h»

«

Si veda la sezione 88.112.

```
3780001 #ifndef _STDIO_H
3780002 #define _STDIO_H        1
3780003 //-----------------------------------------------------------
3780004 #include <restrict.h>
3780005 #include <stdarg.h>
3780006 #include <stdint.h>
3780007 #include <limits.h>
3780008 #include <NULL.h>
3780009 #include <size_t.h>
3780010 #include <sys/types.h>
3780011 #include <SEEK.h>       // SEEK_CUR, SEEK_SET,
3780012                        // SEEK_END
3780013 //-----------------------------------------------------------
3780014 #define BUFSIZ              8192 // At least the
3780015                                  // file
3780016                                  // system max zone
3780017                                  // size.
3780018 #define _IOFBF                 0 // Input-output
3780019                                  // fully
3780020                                  // buffered.
3780021 #define _IOLBF                 1 // Input-output
3780022                                  // line
3780023                                  // buffered.
3780024 #define _IONBF                 2 // Input-output
3780025                                  // with
3780026                                  // no buffering.
3780027
```



```
3780028 #define L_tmpnam    FILENAME_MAX // <limits.h>
3780029
3780030 #define FOPEN_MAX       OPEN_MAX // <limits.h>
3780031 #define FILENAME_MAX    NAME_MAX // <limits.h>
3780032 #define TMP_MAX           0x7FFF
3780033
3780034 #define EOF                (-1) // Must be a
3780035                                  // negative
3780036                                  // value.
3780037 //-----------------------------------------------------------
3780038 typedef off_t fpos_t;   // 'off_t' defined in
3780039                         // <sys/types.h>.
3780040
3780041 typedef struct
3780042 {
3780043   int fdn;      // File descriptor number.
3780044   char error;   // Error indicator.
3780045   char eof;     // End of file indicator.
3780046 } FILE;
3780047
3780048 extern FILE _stream[];  // Defined inside
3780049                         // 'lib/stdio/FILE.c'.
3780050
3780051 #define stdin   (&_stream[0])
3780052 #define stdout  (&_stream[1])
3780053 #define stderr  (&_stream[2])
3780054 //-----------------------------------------------------------
3780055 void clearerr (FILE * fp);
3780056 int fclose (FILE * fp);
3780057 int feof (FILE * fp);
3780058 int ferror (FILE * fp);
3780059 int fflush (FILE * fp);
3780060 int fgetc (FILE * fp);
3780061 int fgetpos (FILE * restrict fp, fpos_t * restrict pos);
3780062 char *fgets (char *restrict string, int n,
3780063              FILE * restrict fp);
3780064 int fileno (FILE * fp);
3780065 FILE *fopen (const char *path, const char *mode);
3780066 int fprintf (FILE * fp, char *restrict format, ...);
3780067 int fputc (int c, FILE * fp);
3780068 int fputs (const char *restrict string, FILE * restrict fp);
3780069 size_t fread (void *restrict buffer, size_t size,
3780070               size_t nmemb, FILE * restrict fp);
3780071 FILE *freopen (const char *restrict path,
3780072                const char *restrict mode,
3780073                FILE * restrict fp);
3780074 int fscanf (FILE * restrict fp,
3780075             const char *restrict format, ...);
3780076 int fseek (FILE * fp, long int offset, int whence);
3780077 int fsetpos (FILE * fp, fpos_t * pos);
3780078 long int ftell (FILE * fp);
3780079 off_t ftello (FILE * fp);
3780080 size_t fwrite (const void *restrict buffer,
3780081                size_t size, size_t nmemb,
3780082                FILE * restrict fp);
3780083 #define  getc(p)    (fgetc (p))
3780084 int getchar (void);
3780085 char *gets (char *string);
3780086 void perror (const char *string);
3780087 int printf (const char *restrict format, ...);
3780088 #define  putc(c, p) (fputc ((c), (p)))
3780089 int putchar (int c);
3780090 int puts (const char *string);
3780091 void rewind (FILE * fp);
3780092 int scanf (const char *restrict format, ...);
3780093 void setbuf (FILE * restrict fp, char *restrict buffer);
3780094 int setvbuf (FILE * restrict fp, char *restrict buffer,
3780095              int buf_mode, size_t size);
3780096 int snprintf (char *restrict string, size_t size,
3780097               const char *restrict format, ...);
3780098 int sprintf (char *restrict string,
3780099              const char *restrict format, ...);
3780100 int sscanf (char *restrict string,
3780101             const char *restrict format, ...);
3780102 int vfprintf (FILE * fp, char *restrict format,
3780103               va_list arg);
3780104 int vfscanf (FILE * restrict fp,
3780105              const char *restrict format, va_list arg);
3780106 int vprintf (const char *restrict format, va_list arg);
3780107 int vscanf (const char *restrict format, va_list ap);
3780108 int vsnprintf (char *restrict string, size_t size,
3780109                const char *restrict format, va_list arg);
3780110 int vsprintf (char *restrict string,
3780111               const char *restrict format, va_list arg);
3780112 int vsscanf (const char *string, const char *format,
3780113              va_list ap);
3780114 //-----------------------------------------------------------
```

```
3780115 #endif
```


95.18.1 lib/stdio/FILE.c 796
95.18.2 lib/stdio/clearerr.c 797
95.18.3 lib/stdio/fclose.c 797
95.18.4 lib/stdio/feof.c 797
95.18.5 lib/stdio/ferror.c 797
95.18.6 lib/stdio/fflush.c 798
95.18.7 lib/stdio/fgetc.c 798
95.18.8 lib/stdio/fgetpos.c 798
95.18.9 lib/stdio/fgets.c 798
95.18.10 lib/stdio/fileno.c 799
95.18.11 lib/stdio/fopen.c 799
95.18.12 lib/stdio/fprintf.c 800
95.18.13 lib/stdio/fputc.c 800
95.18.14 lib/stdio/fputs.c 801
95.18.15 lib/stdio/fread.c 801
95.18.16 lib/stdio/freopen.c 801
95.18.17 lib/stdio/fscanf.c 802
95.18.18 lib/stdio/fseek.c 802
95.18.19 lib/stdio/fseeko.c 802
95.18.20 lib/stdio/fsetpos.c 802
95.18.21 lib/stdio/ftell.c 803
95.18.22 lib/stdio/ftello.c 803
95.18.23 lib/stdio/fwrite.c 803
95.18.24 lib/stdio/getchar.c 803
95.18.25 lib/stdio/gets.c 804
95.18.26 lib/stdio/perror.c 804
95.18.27 lib/stdio/printf.c 805
95.18.28 lib/stdio/putchar.c 805
95.18.29 lib/stdio/puts.c 805
95.18.30 lib/stdio/rewind.c 806
95.18.31 lib/stdio/scanf.c 806
95.18.32 lib/stdio/setbuf.c 806
95.18.33 lib/stdio/setvbuf.c 806
95.18.34 lib/stdio/snprintf.c 806
95.18.35 lib/stdio/sprintf.c 806
95.18.36 lib/stdio/sscanf.c 807
95.18.37 lib/stdio/vfprintf.c 807
95.18.38 lib/stdio/vfscanf.c 807
95.18.39 lib/stdio/vfsscanf.c 807
95.18.40 lib/stdio/vprintf.c 827
95.18.41 lib/stdio/vscanf.c 827
95.18.42 lib/stdio/vsnprintf.c 828
95.18.43 lib/stdio/vsprintf.c 843
95.18.44 lib/stdio/vsscanf.c 844

95.18.1 lib/stdio/FILE.c

Si veda la sezione 91.3.

```
3790001 #include <stdio.h>
3790002 //
3790003 // There must be room for at least 'FOPEN_MAX'
3790004 // elements.
3790005 //
3790006 FILE _stream[FOPEN_MAX];
3790007 //-----------------------------------------------------
3790008 void
3790009 _stdio_stream_setup (void)
3790010 {
3790011   _stream[0].fdn = 0;
3790012   _stream[0].error = 0;
3790013   _stream[0].eof = 0;
3790014
3790015   _stream[1].fdn = 1;
3790016   _stream[1].error = 0;
3790017   _stream[1].eof = 0;
3790018
3790019   _stream[2].fdn = 2;
3790020   _stream[2].error = 0;
3790021   _stream[2].eof = 0;
3790022 }
```

95.18.2 lib/stdio/clearerr.c

Si veda la sezione 88.12.

```
3800001 #include <stdio.h>
3800002 //-----------------------------------------------------
3800003 void
3800004 clearerr (FILE * fp)
3800005 {
3800006   if (fp != NULL)
3800007     {
3800008       fp->error = 0;
3800009       fp->eof = 0;
3800010     }
3800011 }
```

95.18.3 lib/stdio/fclose.c

Si veda la sezione 88.28.

```
3810001 #include <stdio.h>
3810002 #include <unistd.h>
3810003 //-----------------------------------------------------
3810004 int
3810005 fclose (FILE * fp)
3810006 {
3810007   return (close (fp->fdn));
3810008 }
```

95.18.4 lib/stdio/feof.c

Si veda la sezione 88.29.

```
3820001 #include <stdio.h>
3820002 //-----------------------------------------------------
3820003 int
3820004 feof (FILE * fp)
3820005 {
3820006   if (fp != NULL)
3820007     {
3820008       return (fp->eof);
3820009     }
3820010   return (0);
3820011 }
```

95.18.5 lib/stdio/ferror.c

Si veda la sezione 88.30.

```
3830001 #include <stdio.h>
3830002 //-----------------------------------------------------
3830003 int
3830004 ferror (FILE * fp)
3830005 {
3830006   if (fp != NULL)
3830007     {
3830008       return (fp->error);
3830009     }
3830010   return (0);
3830011 }
```

95.18.6 lib/stdio/fflush.c

Si veda la sezione 88.31.

```
3840001 #include <stdio.h>
3840002 //-------------------------------------------------------------
3840003 int
3840004 fflush (FILE * fp)
3840005 {
3840006   //
3840007   // The os32 library does not have any buffered data.
3840008   //
3840009   return (0);
3840010 }
```

95.18.7 lib/stdio/fgetc.c

Si veda la sezione 88.32.

```
3850001 #include <stdio.h>
3850002 #include <sys/types.h>
3850003 #include <unistd.h>
3850004 //-------------------------------------------------------------
3850005 int
3850006 fgetc (FILE * fp)
3850007 {
3850008   ssize_t size_read;
3850009   int c;        // Character read.
3850010   //
3850011   for (c = 0;;)
3850012     {
3850013       size_read = read (fp->fdn, &c, (size_t) 1);
3850014       //
3850015       if (size_read <= 0)
3850016         {
3850017           //
3850018           // It is the end of file (zero) otherwise
3850019           // there is a
3850020           // problem (a negative value): return 'EOF'.
3850021           //
3850022           return (EOF);
3850023         }
3850024       //
3850025       // Valid read: end of scan.
3850026       //
3850027       return (c);
3850028     }
3850029 }
```

95.18.8 lib/stdio/fgetpos.c

Si veda la sezione 88.33.

```
3860001 #include <stdio.h>
3860002 //-------------------------------------------------------------
3860003 int
3860004 fgetpos (FILE * restrict fp, fpos_t * restrict pos)
3860005 {
3860006   long int position;
3860007   //
3860008   if (fp != NULL)
3860009     {
3860010       position = ftell (fp);
3860011       if (position >= 0)
3860012         {
3860013           *pos = position;
3860014           return (0);
3860015         }
3860016     }
3860017   return (-1);
3860018 }
```

95.18.9 lib/stdio/fgets.c

Si veda la sezione 88.34.

```
3870001 #include <stdio.h>
3870002 #include <sys/types.h>
3870003 #include <unistd.h>
3870004 #include <stddef.h>
3870005 //-------------------------------------------------------------
3870006 char *
3870007 fgets (char *restrict string, int n, FILE * restrict fp)
3870008 {
3870009   ssize_t size_read;
3870010   int b;        // Index inside the string buffer.
3870011   //
3870012   for (b = 0; b < (n - 1); b++, string[b] = 0)
3870013     {
3870014       size_read = read (fp->fdn, &string[b], (size_t) 1);
3870015       //
3870016       if (size_read <= 0)
3870017         {
3870018           //
3870019           // It is the end of file (zero) otherwise
3870020           // there is a
3870021           // problem (a negative value).
3870022           //
3870023           string[b] = 0;
3870024           break;
3870025         }
3870026       //
3870027       if (string[b] == '\n')
3870028         {
3870029           b++;
3870030           string[b] = 0;
3870031           break;
3870032         }
3870033     }
3870034   //
3870035   // If 'b' is zero, nothing was read and 'NULL' is
3870036   // returned.
3870037   //
3870038   if (b == 0)
3870039     {
3870040       return (NULL);
3870041     }
3870042   else
3870043     {
3870044       return (string);
3870045     }
3870046 }
```


95.18.10 lib/stdio/fileno.c

Si veda la sezione 88.35.

```
3880001 #include <stdio.h>
3880002 #include <errno.h>
3880003 //-------------------------------------------------------------
3880004 int
3880005 fileno (FILE * fp)
3880006 {
3880007   if (fp != NULL)
3880008     {
3880009       return (fp->fdn);
3880010     }
3880011   errset (EBADF);       // Bad file descriptor.
3880012   return (-1);
3880013 }
```

95.18.11 lib/stdio/fopen.c

Si veda la sezione 88.36.

```
3890001 #include <fcntl.h>
3890002 #include <stdarg.h>
3890003 #include <stddef.h>
3890004 #include <string.h>
3890005 #include <errno.h>
3890006 #include <sys/os32.h>
3890007 #include <limits.h>
3890008 #include <stdio.h>
3890009 //-------------------------------------------------------------
3890010 FILE *
3890011 fopen (const char *path, const char *mode)
3890012 {
3890013   int fdn;
3890014   //
3890015   if (strcmp (mode, "r") || strcmp (mode, "rb"))
3890016     {
3890017       fdn = open (path, O_RDONLY);
3890018     }
3890019   else if (strcmp (mode, "r+") ||
3890020            strcmp (mode, "r+b") || strcmp (mode, "rb+"))
3890021     {
3890022       fdn = open (path, O_RDWR);
3890023     }
3890024   else if (strcmp (mode, "w") || strcmp (mode, "wb"))
3890025     {
3890026       fdn = open (path, O_WRONLY | O_CREAT | O_TRUNC, 0666);
3890027     }
3890028   else if (strcmp (mode, "w+") ||
3890029            strcmp (mode, "w+b") || strcmp (mode, "wb+"))
3890030     {
```

```
3890031       fdn = open (path, O_RDWR | O_CREAT | O_TRUNC, 0666);
3890032     }
3890033   else if (strcmp (mode, "a") || strcmp (mode, "ab"))
3890034     {
3890035       fdn =
3890036         open (path,
3890037               O_WRONLY | O_APPEND | O_CREAT | O_TRUNC,
3890038               0666);
3890039     }
3890040   else if (strcmp (mode, "a+") ||
3890041            strcmp (mode, "a+b") || strcmp (mode, "ab+"))
3890042     {
3890043       fdn =
3890044         open (path,
3890045               O_RDWR | O_APPEND | O_CREAT | O_TRUNC, 0666);
3890046     }
3890047   else
3890048     {
3890049       errset (EINVAL);  // Invalid argument.
3890050       return (NULL);
3890051     }
3890052   //
3890053   // Check the file descriptor returned.
3890054   //
3890055   if (fdn < 0)
3890056     {
3890057       //
3890058       // The variable 'errno' is already set.
3890059       //
3890060       errset (errno);
3890061       return (NULL);
3890062     }
3890063   //
3890064   // A valid file descriptor is available: convert it
3890065   // into a file
3890066   // stream. Please note that the file descriptor
3890067   // number must be
3890068   // saved inside the corresponding '_stream[]' array,
3890069   // because the
3890070   // file pointer do not have knowledge of the
3890071   // relative position
3890072   // inside the array.
3890073   //
3890074   _stream[fdn].fdn = fdn;       // Saved the file
3890075   // descriptor number.
3890076   //
3890077   return (&_stream[fdn]);       // Returned the file
3890078   // stream pointer.
3890079 }
```

95.18.12 lib/stdio/fprintf.c

«

Si veda la sezione 88.91.

```
3900001 #include <stdio.h>
3900002 //-------------------------------------------------------
3900003 int
3900004 fprintf (FILE * fp, char *restrict format, ...)
3900005 {
3900006   va_list ap;
3900007   va_start (ap, format);
3900008   return (vfprintf (fp, format, ap));
3900009 }
```

95.18.13 lib/stdio/fputc.c

«

Si veda la sezione 88.38.

```
3910001 #include <stdio.h>
3910002 #include <sys/types.h>
3910003 #include <sys/os32.h>
3910004 #include <string.h>
3910005 #include <unistd.h>
3910006 //-------------------------------------------------------
3910007 int
3910008 fputc (int c, FILE * fp)
3910009 {
3910010   ssize_t size_written;
3910011   char character = (char) c;
3910012   size_written = write (fp->fdn, &character, (size_t) 1);
3910013   if (size_written < 0)
3910014     {
3910015       fp->eof = 1;
3910016       return (EOF);
3910017     }
3910018   return (c);
3910019 }
```


95.18.14 lib/stdio/fputs.c

«

Si veda la sezione 88.39.

```
3920001 #include <stdio.h>
3920002 #include <string.h>
3920003 //-------------------------------------------------------
3920004 int
3920005 fputs (const char *restrict string, FILE * restrict fp)
3920006 {
3920007   int i;        // Index inside the string to be
3920008   // printed.
3920009   int status;
3920010
3920011   for (i = 0; i < strlen (string); i++)
3920012     {
3920013       status = fputc (string[i], fp);
3920014       if (status == EOF)
3920015         {
3920016           fp->eof = 1;
3920017           return (EOF);
3920018         }
3920019     }
3920020   return (0);
3920021 }
```

95.18.15 lib/stdio/fread.c

«

Si veda la sezione 88.40.

```
3930001 #include <unistd.h>
3930002 #include <stdio.h>
3930003 //-------------------------------------------------------
3930004 size_t
3930005 fread (void *restrict buffer, size_t size,
3930006        size_t nmemb, FILE * restrict fp)
3930007 {
3930008   ssize_t size_read;
3930009   size_read =
3930010     read (fp->fdn, buffer, (size_t) (size * nmemb));
3930011   if (size_read == 0)
3930012     {
3930013       fp->eof = 1;
3930014       return ((size_t) 0);
3930015     }
3930016   else if (size_read < 0)
3930017     {
3930018       fp->error = 1;
3930019       return ((size_t) 0);
3930020     }
3930021   else
3930022     {
3930023       return ((size_t) (size_read / size));
3930024     }
3930025 }
```

95.18.16 lib/stdio/freopen.c

«

Si veda la sezione 88.36.

```
3940001 #include <fcntl.h>
3940002 #include <stdarg.h>
3940003 #include <stddef.h>
3940004 #include <string.h>
3940005 #include <errno.h>
3940006 #include <sys/os32.h>
3940007 #include <limits.h>
3940008 #include <stdio.h>
3940009 //-------------------------------------------------------
3940010 FILE *
3940011 freopen (const char *restrict path,
3940012          const char *restrict mode, FILE * restrict fp)
3940013 {
3940014   int status;
3940015   FILE *fp_new;
3940016   //
3940017   if (fp == NULL)
3940018     {
3940019       return (NULL);
3940020     }
3940021   //
3940022   status = fclose (fp);
3940023   if (status != 0)
3940024     {
3940025       fp->error = 1;
3940026       return (NULL);
3940027     }
3940028   //
```

```
3940029   fp_new = fopen (path, mode);
3940030   //
3940031   if (fp_new == NULL)
3940032     {
3940033       return (NULL);
3940034     }
3940035   //
3940036   if (fp_new != fp)
3940037     {
3940038       fclose (fp_new);
3940039       return (NULL);
3940040     }
3940041   //
3940042   return (fp_new);
3940043 }
```

95.18.17 lib/stdio/fscanf.c

«

Si veda la sezione 88.102.

```
3950001 #include <stdio.h>
3950002 //-----------------------------------------------------------
3950003 int
3950004 fscanf (FILE * restrict fp,
3950005         const char *restrict format, ...)
3950006 {
3950007   va_list ap;
3950008   va_start (ap, format);
3950009   return vfscanf (fp, format, ap);
3950010 }
```

95.18.18 lib/stdio/fseek.c

«

Si veda la sezione 88.44.

```
3960001 #include <stdio.h>
3960002 #include <unistd.h>
3960003 //-----------------------------------------------------------
3960004 int
3960005 fseek (FILE * fp, long int offset, int whence)
3960006 {
3960007   off_t off_new;
3960008   off_new = lseek (fp->fdn, (off_t) offset, whence);
3960009   if (off_new < 0)
3960010     {
3960011       fp->error = 1;
3960012       return (-1);
3960013     }
3960014   else
3960015     {
3960016       fp->eof = 0;
3960017       return (0);
3960018     }
3960019 }
```

95.18.19 lib/stdio/fseeko.c

«

Si veda la sezione 88.44.

```
3970001 #include <stdio.h>
3970002 #include <unistd.h>
3970003 //-----------------------------------------------------------
3970004 int
3970005 fseeko (FILE * fp, off_t offset, int whence)
3970006 {
3970007   off_t off_new;
3970008   off_new = lseek (fp->fdn, offset, whence);
3970009   if (off_new < 0)
3970010     {
3970011       fp->error = 1;
3970012       return (-1);
3970013     }
3970014   else
3970015     {
3970016       return (0);
3970017     }
3970018 }
```

95.18.20 lib/stdio/fsetpos.c

«

Si veda la sezione 88.33.

```
3980001 #include <stdio.h>
3980002 //-----------------------------------------------------------
3980003 int
3980004 fsetpos (FILE * fp, fpos_t * pos)
```



```
3980005 {
3980006   long int position;
3980007   //
3980008   if (fp != NULL)
3980009     {
3980010       position = fseek (fp, (long int) *pos, SEEK_SET);
3980011       if (position >= 0)
3980012         {
3980013           *pos = position;
3980014           return (0);
3980015         }
3980016     }
3980017   return (-1);
3980018 }
```

95.18.21 lib/stdio/ftell.c

«

Si veda la sezione 88.47.

```
3990001 #include <stdio.h>
3990002 #include <unistd.h>
3990003 //-----------------------------------------------------------
3990004 long int
3990005 ftell (FILE * fp)
3990006 {
3990007   return ((long int) lseek (fp->fdn, (off_t) 0, SEEK_CUR));
3990008 }
```

95.18.22 lib/stdio/ftello.c

«

Si veda la sezione 88.47.

```
4000001 #include <stdio.h>
4000002 #include <unistd.h>
4000003 //-----------------------------------------------------------
4000004 off_t
4000005 ftello (FILE * fp)
4000006 {
4000007   return (lseek (fp->fdn, (off_t) 0, SEEK_CUR));
4000008 }
```

95.18.23 lib/stdio/fwrite.c

«

Si veda la sezione 88.49.

```
4010001 #include <unistd.h>
4010002 #include <stdio.h>
4010003 //-----------------------------------------------------------
4010004 size_t
4010005 fwrite (const void *restrict buffer, size_t size,
4010006         size_t nmemb, FILE * restrict fp)
4010007 {
4010008   ssize_t size_written;
4010009   size_written =
4010010     write (fp->fdn, buffer, (size_t) (size * nmemb));
4010011   if (size_written < 0)
4010012     {
4010013       fp->error = 1;
4010014       return ((size_t) 0);
4010015     }
4010016   else
4010017     {
4010018       return ((size_t) (size_written / size));
4010019     }
4010020 }
```

95.18.24 lib/stdio/getchar.c

«

Si veda la sezione 88.32.

```
4020001 #include <stdio.h>
4020002 #include <sys/types.h>
4020003 #include <unistd.h>
4020004 //-----------------------------------------------------------
4020005 int
4020006 getchar (void)
4020007 {
4020008   ssize_t size_read;
4020009   int c;        // Character read.
4020010   //
4020011   for (c = 0;;)
4020012     {
4020013       size_read = read (STDIN_FILENO, &c, (size_t) 1);
4020014       //
4020015       if (size_read <= 0)
4020016         {
```

```
4020017             //
4020018             // It is the end of file (zero) otherwise
4020019             // there is a
4020020             // problem (a negative value): return 'EOF'.
4020021             //
4020022             _stream[STDIN_FILENO].eof = 1;
4020023             return (EOF);
4020024           }
4020025         //
4020026         // Valid read.
4020027         //
4020028         if (size_read == 0)
4020029           {
4020030             //
4020031             // If no character is ready inside the
4020032             // keyboard buffer, just
4020033             // retry.
4020034             //
4020035             continue;
4020036           }
4020037         //
4020038         // End of scan.
4020039         //
4020040         return (c);
4020041       }
4020042 }
```

95.18.25 lib/stdio/gets.c

Si veda la sezione 88.34.

```
4030001 #include <stdio.h>
4030002 #include <sys/types.h>
4030003 #include <unistd.h>
4030004 #include <stddef.h>
4030005 //-------------------------------------------------------
4030006 char *
4030007 gets (char *string)
4030008 {
4030009   ssize_t size_read;
4030010   int b;        // Index inside the string buffer.
4030011   //
4030012   for (b = 0;; b++, string[b] = 0)
4030013     {
4030014       size_read =
4030015         read (STDIN_FILENO, &string[b], (size_t) 1);
4030016       //
4030017       if (size_read <= 0)
4030018         {
4030019           //
4030020           // It is the end of file (zero) otherwise
4030021           // there is a
4030022           // problem (a negative value).
4030023           //
4030024           _stream[STDIN_FILENO].eof = 1;
4030025           string[b] = 0;
4030026           break;
4030027         }
4030028       //
4030029       if (string[b] == '\n')
4030030         {
4030031           b++;
4030032           string[b] = 0;
4030033           break;
4030034         }
4030035     }
4030036   //
4030037   // If 'b' is zero, nothing was read and 'NULL' is
4030038   // returned.
4030039   //
4030040   if (b == 0)
4030041     {
4030042       return (NULL);
4030043     }
4030044   else
4030045     {
4030046       return (string);
4030047     }
4030048 }
```

95.18.26 lib/stdio/perror.c

Si veda la sezione 88.90.

```
4040001 #include <stdio.h>
4040002 #include <errno.h>
4040003 #include <stddef.h>
```



```
4040004 #include <string.h>
4040005 //-------------------------------------------------------
4040006 void
4040007 perror (const char *string)
4040008 {
4040009   //
4040010   // If errno is zero, there is nothing to show.
4040011   //
4040012   if (errno == 0)
4040013     {
4040014       return;
4040015     }
4040016   //
4040017   // Show the string if there is one.
4040018   //
4040019   if (string != NULL && strlen (string) > 0)
4040020     {
4040021       printf ("%s: ", string);
4040022     }
4040023   //
4040024   // Show the translated error.
4040025   //
4040026   if (errfn[0] != 0 && errln != 0)
4040027     {
4040028       printf ("[%s:%u:%i] %s\n",
4040029               errfn, errln, errno, strerror (errno));
4040030     }
4040031   else
4040032     {
4040033       printf ("[%i] %s\n", errno, strerror (errno));
4040034     }
4040035 }
```

95.18.27 lib/stdio/printf.c

Si veda la sezione 88.91.

```
4050001 #include <stdio.h>
4050002 //-------------------------------------------------------
4050003 int
4050004 printf (const char *restrict format, ...)
4050005 {
4050006   va_list ap;
4050007   va_start (ap, format);
4050008   return (vprintf (format, ap));
4050009 }
```

95.18.28 lib/stdio/putchar.c

Si veda la sezione 88.38.

```
4060001 #include <stdio.h>
4060002 #include <sys/types.h>
4060003 #include <sys/os32.h>
4060004 #include <string.h>
4060005 #include <unistd.h>
4060006 //-------------------------------------------------------
4060007 int
4060008 putchar (int c)
4060009 {
4060010   return (fputc (c, stdout));
4060011 }
```

95.18.29 lib/stdio/puts.c

Si veda la sezione 88.39.

```
4070001 #include <stdio.h>
4070002 //-------------------------------------------------------
4070003 int
4070004 puts (const char *string)
4070005 {
4070006   int status;
4070007   status = printf ("%s\n", string);
4070008   if (status < 0)
4070009     {
4070010       return (EOF);
4070011     }
4070012   else
4070013     {
4070014       return (status);
4070015     }
4070016 }
```

95.18.30 lib/stdio/rewind.c

Si veda la sezione 88.100.

```
4080001 #include <stdio.h>
4080002 //-------------------------------------------------------
4080003 void
4080004 rewind (FILE * fp)
4080005 {
4080006   (void) fseek (fp, 0L, SEEK_SET);
4080007   fp->error = 0;
4080008 }
```

95.18.31 lib/stdio/scanf.c

Si veda la sezione 88.102.

```
4090001 #include <stdio.h>
4090002 //-------------------------------------------------------
4090003 int
4090004 scanf (const char *restrict format, ...)
4090005 {
4090006   va_list ap;
4090007   va_start (ap, format);
4090008   return vfscanf (stdin, format, ap);
4090009 }
```

95.18.32 lib/stdio/setbuf.c

Si veda la sezione 88.103.

```
4100001 #include <stdio.h>
4100002 //-------------------------------------------------------
4100003 void
4100004 setbuf (FILE * restrict fp, char *restrict buffer)
4100005 {
4100006   //
4100007   // The os32 library does not have any buffered data.
4100008   //
4100009   return;
4100010 }
```

95.18.33 lib/stdio/setvbuf.c

Si veda la sezione 88.103.

```
4110001 #include <stdio.h>
4110002 //-------------------------------------------------------
4110003 int
4110004 setvbuf (FILE * restrict fp, char *restrict buffer,
4110005          int buf_mode, size_t size)
4110006 {
4110007   //
4110008   // The os32 library does not have any buffered data.
4110009   //
4110010   return (0);
4110011 }
```

95.18.34 lib/stdio/snprintf.c

Si veda la sezione 88.91.

```
4120001 #include <stdio.h>
4120002 #include <stdarg.h>
4120003 //-------------------------------------------------------
4120004 int
4120005 snprintf (char *restrict string, size_t size,
4120006           const char *restrict format, ...)
4120007 {
4120008   va_list ap;
4120009   va_start (ap, format);
4120010   return vsnprintf (string, size, format, ap);
4120011 }
```

95.18.35 lib/stdio/sprintf.c

Si veda la sezione 88.91.

```
4130001 #include <stdio.h>
4130002 #include <stdarg.h>
4130003 //-------------------------------------------------------
4130004 int
4130005 sprintf (char *restrict string,
4130006          const char *restrict format, ...)
4130007 {
4130008   va_list ap;
4130009   va_start (ap, format);
```



```
4130010   return vsnprintf (string, (size_t) BUFSIZ, format, ap);
4130011 }
```

95.18.36 lib/stdio/sscanf.c

Si veda la sezione 88.102.

```
4140001 #include <stdio.h>
4140002 //-------------------------------------------------------
4140003 int
4140004 sscanf (char *restrict string,
4140005         const char *restrict format, ...)
4140006 {
4140007   va_list ap;
4140008   va_start (ap, format);
4140009   return vsscanf (string, format, ap);
4140010 }
```

95.18.37 lib/stdio/vfprintf.c

Si veda la sezione 88.137.

```
4150001 #include <stdio.h>
4150002 #include <sys/types.h>
4150003 #include <sys/os32.h>
4150004 #include <string.h>
4150005 #include <unistd.h>
4150006 //-------------------------------------------------------
4150007 int
4150008 vfprintf (FILE * fp, char *restrict format, va_list arg)
4150009 {
4150010   ssize_t size_written;
4150011   size_t size;
4150012   size_t size_total;
4150013   int status;
4150014   char string[BUFSIZ];
4150015   char *buffer = string;
4150016   //
4150017   buffer[0] = 0;
4150018   status = vsprintf (buffer, format, arg);
4150019   //
4150020   size = strlen (buffer);
4150021   if (size >= BUFSIZ)
4150022     {
4150023       size = BUFSIZ;
4150024     }
4150025   //
4150026   for (size_total = 0, size_written = 0;
4150027        size_total < size;
4150028        size_total += size_written, buffer += size_written)
4150029     {
4150030       size_written =
4150031         write (fp->fdn, buffer, size - size_total);
4150032       if (size_written < 0)
4150033         {
4150034           return (size_total);
4150035         }
4150036     }
4150037   return (size);
4150038 }
```

95.18.38 lib/stdio/vfscanf.c

Si veda la sezione 88.138.

```
4160001 #include <stdio.h>
4160002
4160003 //-------------------------------------------------------
4160004 int vfsscanf (FILE * restrict fp, const char *string,
4160005                const char *restrict format, va_list ap);
4160006 //-------------------------------------------------------
4160007 int
4160008 vfscanf (FILE * restrict fp,
4160009          const char *restrict format, va_list ap)
4160010 {
4160011   return (vfsscanf (fp, NULL, format, ap));
4160012 }
4160013
4160014 //-------------------------------------------------------
```

95.18.39 lib/stdio/vfsscanf.c

Si veda la sezione 88.138.

```
4170001 #include <stdint.h>
4170002 #include <stdbool.h>
```

```
4170003 #include <stdlib.h>
4170004 #include <string.h>
4170005 #include <stdio.h>
4170006 #include <stdarg.h>
4170007 #include <ctype.h>
4170008 #include <errno.h>
4170009 #include <stddef.h>
4170010 //-------------------------------------------------------
4170011 //
4170012 // This function is not standard and is able to do the
4170013 // work of both 'vfscanf()' and 'vsscanf()'.
4170014 //
4170015 //-------------------------------------------------------
4170016 #define WIDTH_MAX       64
4170017 //-------------------------------------------------------
4170018 static intmax_t strtointmax (const char *restrict
4170019                              string,
4170020                              const char **restrict
4170021                              endptr, int base,
4170022                              size_t max_width);
4170023 static int ass_or_eof (int consumed, int assigned);
4170024 //-------------------------------------------------------
4170025 int
4170026 vfsscanf (FILE * restrict fp, const char *string,
4170027           const char *restrict format, va_list ap)
4170028 {
4170029   int f = 0;    // Format index.
4170030   char buffer[BUFSIZ];
4170031   const char *input = string;   // Default.
4170032   const char *start = input;    // Default.
4170033   const char *restrict next = NULL;
4170034   int scanned = 0;
4170035   //
4170036   bool stream = 0;
4170037   bool flag_star = 0;
4170038   bool specifier = 0;
4170039   bool specifier_flags = 0;
4170040   bool specifier_width = 0;
4170041   bool specifier_type = 0;
4170042   bool inverted = 0;
4170043   //
4170044   char *ptr_char;
4170045   signed char *ptr_schar;
4170046   unsigned char *ptr_uchar;
4170047   short int *ptr_sshort;
4170048   unsigned short int *ptr_ushort;
4170049   int *ptr_sint;
4170050   unsigned int *ptr_uint;
4170051   long int *ptr_slong;
4170052   unsigned long int *ptr_ulong;
4170053   intmax_t *ptr_simax;
4170054   uintmax_t *ptr_uimax;
4170055   size_t *ptr_size;
4170056   ptrdiff_t *ptr_ptrdiff;
4170057   void **ptr_void;
4170058   //
4170059   size_t width;
4170060   char width_string[WIDTH_MAX + 1];
4170061   int w;        // Index inside width string.
4170062   int assigned = 0;     // Assignment counter.
4170063   int consumed = 0;     // Consumed counter.
4170064   //
4170065   intmax_t value_i;
4170066   uintmax_t value_u;
4170067   //
4170068   const char *end_format;
4170069   const char *end_input;
4170070   int count;    // Generic counter.
4170071   int index;    // Generic index.
4170072   bool ascii[128];
4170073   //
4170074   void *pstatus;
4170075   //
4170076   // Initialize some data.
4170077   //
4170078   width_string[0] = '\0';
4170079   end_format = format + (strlen (format));
4170080   //
4170081   // Check arguments and find where input comes.
4170082   //
4170083   if (fp == NULL && (string == NULL || string[0] == 0))
4170084     {
4170085       errset (EINVAL);  // Invalid argument.
4170086       return (EOF);
4170087     }
4170088   //
4170089   if (fp != NULL && string != NULL && string[0] != 0)
4170090     {
4170091       errset (EINVAL);  // Invalid argument.
4170092       return (EOF);
4170093     }
4170094   //
4170095   if (fp != NULL)
4170096     {
4170097       stream = 1;
4170098     }
4170099   //
4170100   //
4170101   //
4170102   for (;;)
4170103     {
4170104       if (stream)
4170105         {
4170106           pstatus = fgets (buffer, BUFSIZ, fp);
4170107           //
4170108           if (pstatus == NULL)
4170109             {
4170110               return (ass_or_eof (consumed, assigned));
4170111             }
4170112           //
4170113           input = buffer;
4170114           start = input;
4170115           next = NULL;
4170116         }
4170117       //
4170118       // Calculate end input.
4170119       //
4170120       end_input = input + (strlen (input));
4170121       //
4170122       // Scan format and input strings. Index 'f' is
4170123       // not reset.
4170124       //
4170125       while (&format[f] < end_format && input < end_input)
4170126         {
4170127           if (!specifier)
4170128             {
4170129               // ------------------------------------
4170130               // The context is not
4170131               // inside a specifier.
4170132               // ------------------------------------
4170133               if (isspace (format[f]))
4170134                 {
4170135                   // -------------------------- Space.
4170136                   while (isspace (*input))
4170137                     {
4170138                       input++;
4170139                     }
4170140                   //
4170141                   // Verify that the input string is
4170142                   // not finished.
4170143                   //
4170144                   if (input[0] == 0)
4170145                     {
4170146                       //
4170147                       // As the input string is
4170148                       // finished, the format
4170149                       // string index is not advanced,
4170150                       // because there
4170151                       // might be more spaces on the
4170152                       // next line (if
4170153                       // there is a next line, of
4170154                       // course).
4170155                       //
4170156                       continue;
4170157                     }
4170158                   else
4170159                     {
4170160                       f++;
4170161                       continue;
4170162                     }
4170163                 }
4170164               if (format[f] != '%')
4170165                 {
4170166                   // -------------- Ordinary character.
4170167                   if (format[f] == *input)
4170168                     {
4170169                       input++;
4170170                       f++;
4170171                       continue;
4170172                     }
4170173                   else
4170174                     {
4170175                       return (ass_or_eof
4170176                               (consumed, assigned));
```

```
4170177                     }
4170178                 }
4170179               if (format[f] == '%' && format[f + 1] == '%')
4170180                 {
4170181                   // ---------- Matching a literal '%'.
4170182                   f++;
4170183                   if (format[f] == *input)
4170184                     {
4170185                       input++;
4170186                       f++;
4170187                       continue;
4170188                     }
4170189                   else
4170190                     {
4170191                       return (ass_or_eof
4170192                               (consumed, assigned));
4170193                     }
4170194                 }
4170195               if (format[f] == '%')
4170196                 {
4170197                   // ---------- Percent of a specifier.
4170198                   f++;
4170199                   specifier = 1;
4170200                   specifier_flags = 1;
4170201                   continue;
4170202                 }
4170203             }
4170204           //
4170205           if (specifier && specifier_flags)
4170206             {
4170207               // -------------------------------------
4170208               // The context is inside
4170209               // specifier flags.
4170210               // -------------------------------------
4170211               if (format[f] == '*')
4170212                 {
4170213                   // ----- Assignment suppression star.
4170214                   flag_star = 1;
4170215                   f++;
4170216                 }
4170217               else
4170218                 {
4170219                   // --------------------------------
4170220                   // End of flags and begin of
4170221                   // specifier length.
4170222                   // --------------------------------
4170223                   specifier_flags = 0;
4170224                   specifier_width = 1;
4170225                 }
4170226             }
4170227           //
4170228           if (specifier && specifier_width)
4170229             {
4170230               // -------------------------------------
4170231               // The context is inside a
4170232               // specifier width.
4170233               // -------------------------------------
4170234               for (w = 0;
4170235                    format[f] >= '0'
4170236                    && format[f] <= '9'
4170237                    && w < WIDTH_MAX; w++)
4170238                 {
4170239                   width_string[w] = format[f];
4170240                   f++;
4170241                 }
4170242               width_string[w] = '\0';
4170243               width = atoi (width_string);
4170244               if (width > WIDTH_MAX)
4170245                 {
4170246                   width = WIDTH_MAX;
4170247                 }
4170248               //
4170249               // -------------------------------------
4170250               // A zero width means an unspecified
4170251               // limit for the field
4170252               // length.
4170253               // -------------------------------------
4170254               // End of spec. width and
4170255               // begin of spec. type.
4170256               // -------------------------------------
4170257               specifier_width = 0;
4170258               specifier_type = 1;
4170259             }
4170260           //
4170261           if (specifier && specifier_type)
4170262             {
4170263               //
```



```
4170264               // Specifiers with length modifier.
4170265               //
4170266               if (format[f] == 'h' && format[f + 1] == 'h')
4170267                 {
4170268                   // ---------------------------- char.
4170269                   if (format[f + 2] == 'd')
4170270                     {
4170271                       // -------- signed char, base 10.
4170272                       value_i =
4170273                         strtointmax (input, &next, 10,
4170274                                      width);
4170275                       if (input == next)
4170276                         {
4170277                           return (ass_or_eof
4170278                                   (consumed, assigned));
4170279                         }
4170280                       consumed++;
4170281                       if (!flag_star)
4170282                         {
4170283                           ptr_schar =
4170284                             va_arg (ap, signed char *);
4170285                           *ptr_schar = value_i;
4170286                           assigned++;
4170287                         }
4170288                       f += 3;
4170289                       input = next;
4170290                     }
4170291                   else if (format[f + 2] == 'i')
4170292                     {
4170293                       // ----------------------------
4170294                       // signed char, base unknown.
4170295                       // ----------------------------
4170296                       value_i =
4170297                         strtointmax (input, &next, 0,
4170298                                      width);
4170299                       if (input == next)
4170300                         {
4170301                           return (ass_or_eof
4170302                                   (consumed, assigned));
4170303                         }
4170304                       consumed++;
4170305                       if (!flag_star)
4170306                         {
4170307                           ptr_schar =
4170308                             va_arg (ap, signed char *);
4170309                           *ptr_schar = value_i;
4170310                           assigned++;
4170311                         }
4170312                       f += 3;
4170313                       input = next;
4170314                     }
4170315                   else if (format[f + 2] == 'o')
4170316                     {
4170317                       // ----------------------------
4170318                       // signed char, base 8.
4170319                       // ----------------------------
4170320                       value_i =
4170321                         strtointmax (input, &next, 8,
4170322                                      width);
4170323                       if (input == next)
4170324                         {
4170325                           return (ass_or_eof
4170326                                   (consumed, assigned));
4170327                         }
4170328                       consumed++;
4170329                       if (!flag_star)
4170330                         {
4170331                           ptr_schar =
4170332                             va_arg (ap, signed char *);
4170333                           *ptr_schar = value_i;
4170334                           assigned++;
4170335                         }
4170336                       f += 3;
4170337                       input = next;
4170338                     }
4170339                   else if (format[f + 2] == 'u')
4170340                     {
4170341                       // ----------------------------
4170342                       // unsigned char, base 10.
4170343                       // ----------------------------
4170344                       value_u =
4170345                         strtointmax (input, &next, 10,
4170346                                      width);
4170347                       if (input == next)
4170348                         {
4170349                           return (ass_or_eof
4170350                                   (consumed, assigned));
```

```
                    }
                  consumed++;
                  if (!flag_star)
                    {
                      ptr_uchar =
                        va_arg (ap, unsigned char *);
                      *ptr_uchar = value_u;
                      assigned++;
                    }
                  f += 3;
                  input = next;
                }
              else if (format[f + 2] == 'x'
                       || format[f + 2] == 'X')
                {
                  // ----------------------------
                  // signed char, base 16.
                  // ----------------------------
                  value_i =
                    strtointmax (input, &next, 16,
                                 width);
                  if (input == next)
                    {
                      return (ass_or_eof
                              (consumed, assigned));
                    }
                  consumed++;
                  if (!flag_star)
                    {
                      ptr_schar =
                        va_arg (ap, signed char *);
                      *ptr_schar = value_i;
                      assigned++;
                    }
                  f += 3;
                  input = next;
                }
              else if (format[f + 2] == 'n')
                {
                  // ----------------------------
                  // signed char,
                  // string index counter.
                  // ----------------------------
                  ptr_schar =
                    va_arg (ap, signed char *);
                  *ptr_schar =
                    (signed char) (input - start +
                                   scanned);
                  f += 3;
                }
              else
                {
                  // ----------------------------
                  // unsupported or
                  // unknown specifier.
                  // ----------------------------
                  f += 2;
                }
            }
          else if (format[f] == 'h')
            {
              // -------------------------- short.
              if (format[f + 1] == 'd')
                {
                  // ----------------------------
                  // signed short, base 10.
                  // ----------------------------
                  value_i =
                    strtointmax (input, &next, 10,
                                 width);
                  if (input == next)
                    {
                      return (ass_or_eof
                              (consumed, assigned));
                    }
                  consumed++;
                  if (!flag_star)
                    {
                      ptr_sshort =
                        va_arg (ap, signed short *);
                      *ptr_sshort = value_i;
                      assigned++;
                    }
                  f += 2;
                  input = next;
                }
              else if (format[f + 1] == 'i')
```



```
                {
                  // ----------------------------
                  // signed
                  // short, base unknown.
                  // ----------------------------
                  value_i =
                    strtointmax (input, &next, 0,
                                 width);
                  if (input == next)
                    {
                      return (ass_or_eof
                              (consumed, assigned));
                    }
                  consumed++;
                  if (!flag_star)
                    {
                      ptr_sshort =
                        va_arg (ap, signed short *);
                      *ptr_sshort = value_i;
                      assigned++;
                    }
                  f += 2;
                  input = next;
                }
              else if (format[f + 1] == 'o')
                {
                  // ----------------------------
                  // signed short, base 8.
                  // ----------------------------
                  value_i =
                    strtointmax (input, &next, 8,
                                 width);
                  if (input == next)
                    {
                      return (ass_or_eof
                              (consumed, assigned));
                    }
                  consumed++;
                  if (!flag_star)
                    {
                      ptr_sshort =
                        va_arg (ap, signed short *);
                      *ptr_sshort = value_i;
                      assigned++;
                    }
                  f += 2;
                  input = next;
                }
              else if (format[f + 1] == 'u')
                {
                  // ----------------------------
                  // unsigned short, base 10.
                  // ----------------------------
                  value_u =
                    strtointmax (input, &next, 10,
                                 width);
                  if (input == next)
                    {
                      return (ass_or_eof
                              (consumed, assigned));
                    }
                  consumed++;
                  if (!flag_star)
                    {
                      ptr_ushort =
                        va_arg (ap, unsigned short *);
                      *ptr_ushort = value_u;
                      assigned++;
                    }
                  f += 2;
                  input = next;
                }
              else if (format[f + 1] == 'x'
                       || format[f + 2] == 'X')
                {
                  // ----------------------------
                  // signed short, base 16.
                  // ----------------------------
                  value_i =
                    strtointmax (input, &next, 16,
                                 width);
                  if (input == next)
                    {
                      return (ass_or_eof
                              (consumed, assigned));
                    }
                  consumed++;
```

```
                    if (!flag_star)
                      {
                        ptr_sshort =
                          va_arg (ap, signed short *);
                        *ptr_sshort = value_i;
                        assigned++;
                      }
                    f += 2;
                    input = next;
                  }
                else if (format[f + 1] == 'n')
                  {
                    // ----------------------------
                    // signed char,
                    // string index counter.
                    // ----------------------------
                    ptr_sshort =
                      va_arg (ap, signed short *);
                    *ptr_sshort =
                      (signed short) (input - start +
                                      scanned);
                    f += 2;
                  }
                else
                  {
                    // ----------------------------
                    // unsupported or
                    // unknown specifier.
                    // ----------------------------
                    f += 1;
                  }
              }
            // --------- There is no 'long long int'.
            else if (format[f] == 'l')
              {
                // ---------------------- long int.
                if (format[f + 1] == 'd')
                  {
                    // ----------------------------
                    // signed long, base 10.
                    // ----------------------------
                    value_i =
                      strtointmax (input, &next, 10,
                                   width);
                    if (input == next)
                      {
                        return (ass_or_eof
                                (consumed, assigned));
                      }
                    consumed++;
                    if (!flag_star)
                      {
                        ptr_slong =
                          va_arg (ap, signed long *);
                        *ptr_slong = value_i;
                        assigned++;
                      }
                    f += 2;
                    input = next;
                  }
                else if (format[f + 1] == 'i')
                  {
                    // ----------------------------
                    // signed
                    // long, base unknown.
                    // ----------------------------
                    value_i =
                      strtointmax (input, &next, 0,
                                   width);
                    if (input == next)
                      {
                        return (ass_or_eof
                                (consumed, assigned));
                      }
                    consumed++;
                    if (!flag_star)
                      {
                        ptr_slong =
                          va_arg (ap, signed long *);
                        *ptr_slong = value_i;
                        assigned++;
                      }
                    f += 2;
                    input = next;
                  }
                else if (format[f + 1] == 'o')
                  {
```



```
                    // ----------------------------
                    // signed long, base 8.
                    // ----------------------------
                    value_i =
                      strtointmax (input, &next, 8,
                                   width);
                    if (input == next)
                      {
                        return (ass_or_eof
                                (consumed, assigned));
                      }
                    consumed++;
                    if (!flag_star)
                      {
                        ptr_slong =
                          va_arg (ap, signed long *);
                        *ptr_slong = value_i;
                        assigned++;
                      }
                    f += 2;
                    input = next;
                  }
                else if (format[f + 1] == 'u')
                  {
                    // ----------------------------
                    // unsigned long, base 10.
                    // ----------------------------
                    value_u =
                      strtointmax (input, &next, 10,
                                   width);
                    if (input == next)
                      {
                        return (ass_or_eof
                                (consumed, assigned));
                      }
                    consumed++;
                    if (!flag_star)
                      {
                        ptr_ulong =
                          va_arg (ap, unsigned long *);
                        *ptr_ulong = value_u;
                        assigned++;
                      }
                    f += 2;
                    input = next;
                  }
                else if (format[f + 1] == 'x'
                         || format[f + 2] == 'X')
                  {
                    // ----------------------------
                    // signed long, base 16.
                    // ----------------------------
                    value_i =
                      strtointmax (input, &next, 16,
                                   width);
                    if (input == next)
                      {
                        return (ass_or_eof
                                (consumed, assigned));
                      }
                    consumed++;
                    if (!flag_star)
                      {
                        ptr_slong =
                          va_arg (ap, signed long *);
                        *ptr_slong = value_i;
                        assigned++;
                      }
                    f += 2;
                    input = next;
                  }
                else if (format[f + 1] == 'n')
                  {
                    // ----------------------------
                    // signed char,
                    // string index counter.
                    // ----------------------------
                    ptr_slong =
                      va_arg (ap, signed long *);
                    *ptr_slong =
                      (signed long) (input - start +
                                     scanned);
                    f += 2;
                  }
                else
                  {
                    // ----------------------------
```

```
4170699                     // unsupported or
4170700                     // unknown specifier.
4170701                     // ----------------------------
4170702                     f += 1;
4170703                   }
4170704               }
4170705             else if (format[f] == 'j')
4170706               {
4170707                 // -----------------.------ intmax_t.
4170708                 if (format[f + 1] == 'd')
4170709                   {
4170710                     // ---------- intmax_t, base 10.
4170711                     value_i =
4170712                       strtointmax (input, &next, 10,
4170713                                    width);
4170714                     if (input == next)
4170715                       {
4170716                         return (ass_or_eof
4170717                                 (consumed, assigned));
4170718                       }
4170719                     consumed++;
4170720                     if (!flag_star)
4170721                       {
4170722                         ptr_simax =
4170723                           va_arg (ap, intmax_t *);
4170724                         *ptr_simax = value_i;
4170725                         assigned++;
4170726                       }
4170727                     f += 2;
4170728                     input = next;
4170729                   }
4170730                 else if (format[f + 1] == 'i')
4170731                   {
4170732                     // ----------------------------
4170733                     // intmax_t, base unknown.
4170734                     // ----------------------------
4170735                     value_i =
4170736                       strtointmax (input, &next, 0,
4170737                                    width);
4170738                     if (input == next)
4170739                       {
4170740                         return (ass_or_eof
4170741                                 (consumed, assigned));
4170742                       }
4170743                     consumed++;
4170744                     if (!flag_star)
4170745                       {
4170746                         ptr_simax =
4170747                           va_arg (ap, intmax_t *);
4170748                         *ptr_simax = value_i;
4170749                         assigned++;
4170750                       }
4170751                     f += 2;
4170752                     input = next;
4170753                   }
4170754                 else if (format[f + 1] == 'o')
4170755                   {
4170756                     // ----------------------------
4170757                     // intmax_t, base 8.
4170758                     // ----------------------------
4170759                     value_i =
4170760                       strtointmax (input, &next, 8,
4170761                                    width);
4170762                     if (input == next)
4170763                       {
4170764                         return (ass_or_eof
4170765                                 (consumed, assigned));
4170766                       }
4170767                     consumed++;
4170768                     if (!flag_star)
4170769                       {
4170770                         ptr_simax =
4170771                           va_arg (ap, intmax_t *);
4170772                         *ptr_simax = value_i;
4170773                         assigned++;
4170774                       }
4170775                     f += 2;
4170776                     input = next;
4170777                   }
4170778                 else if (format[f + 1] == 'u')
4170779                   {
4170780                     // ----------------------------
4170781                     // uintmax_t, base 10.
4170782                     // ----------------------------
4170783                     value_u =
4170784                       strtointmax (input, &next, 10,
4170785                                    width);
```

```
4170786                     if (input == next)
4170787                       {
4170788                         return (ass_or_eof
4170789                                 (consumed, assigned));
4170790                       }
4170791                     consumed++;
4170792                     if (!flag_star)
4170793                       {
4170794                         ptr_uimax =
4170795                           va_arg (ap, uintmax_t *);
4170796                         *ptr_uimax = value_u;
4170797                         assigned++;
4170798                       }
4170799                     f += 2;
4170800                     input = next;
4170801                   }
4170802                 else if (format[f + 1] == 'x'
4170803                          || format[f + 2] == 'X')
4170804                   {
4170805                     // ----------------------------
4170806                     // intmax_t, base 16.
4170807                     // ----------------------------
4170808                     value_i =
4170809                       strtointmax (input, &next, 16,
4170810                                    width);
4170811                     if (input == next)
4170812                       {
4170813                         return (ass_or_eof
4170814                                 (consumed, assigned));
4170815                       }
4170816                     consumed++;
4170817                     if (!flag_star)
4170818                       {
4170819                         ptr_simax =
4170820                           va_arg (ap, intmax_t *);
4170821                         *ptr_simax = value_i;
4170822                         assigned++;
4170823                       }
4170824                     f += 2;
4170825                     input = next;
4170826                   }
4170827                 else if (format[f + 1] == 'n')
4170828                   {
4170829                     // ----------------------------
4170830                     // signed char,
4170831                     // string index counter.
4170832                     // ----------------------------
4170833                     ptr_simax = va_arg (ap, intmax_t *);
4170834                     *ptr_simax =
4170835                       (intmax_t) (input - start +
4170836                                   scanned);
4170837                     f += 2;
4170838                   }
4170839                 else
4170840                   {
4170841                     // ----------------------------
4170842                     // unsupported or
4170843                     // unknown specifier.
4170844                     // ----------------------------
4170845                     f += 1;
4170846                   }
4170847               }
4170848             else if (format[f] == 'z')
4170849               {
4170850                 // ------------------------- size_t.
4170851                 if (format[f + 1] == 'd')
4170852                   {
4170853                     // ----------------------------
4170854                     // size_t, base 10.
4170855                     // ----------------------------
4170856                     value_i =
4170857                       strtointmax (input, &next, 10,
4170858                                    width);
4170859                     if (input == next)
4170860                       {
4170861                         return (ass_or_eof
4170862                                 (consumed, assigned));
4170863                       }
4170864                     consumed++;
4170865                     if (!flag_star)
4170866                       {
4170867                         ptr_size = va_arg (ap, size_t *);
4170868                         *ptr_size = value_i;
4170869                         assigned++;
4170870                       }
4170871                     f += 2;
4170872                     input = next;
```

```
4170873                     }
4170874                   else if (format[f + 1] == 'i')
4170875                     {
4170876                       // ----------------------------
4170877                       // size_t, base unknown.
4170878                       // ----------------------------
4170879                       value_i =
4170880                         strtointmax (input, &next, 0,
4170881                                      width);
4170882                       if (input == next)
4170883                         {
4170884                           return (ass_or_eof
4170885                                   (consumed, assigned));
4170886                         }
4170887                       consumed++;
4170888                       if (!flag_star)
4170889                         {
4170890                           ptr_size = va_arg (ap, size_t *);
4170891                           *ptr_size = value_i;
4170892                           assigned++;
4170893                         }
4170894                       f += 2;
4170895                       input = next;
4170896                     }
4170897                   else if (format[f + 1] == 'o')
4170898                     {
4170899                       // ----------------------------
4170900                       // size_t, base 8.
4170901                       // ----------------------------
4170902                       value_i =
4170903                         strtointmax (input, &next, 8,
4170904                                      width);
4170905                       if (input == next)
4170906                         {
4170907                           return (ass_or_eof
4170908                                   (consumed, assigned));
4170909                         }
4170910                       consumed++;
4170911                       if (!flag_star)
4170912                         {
4170913                           ptr_size = va_arg (ap, size_t *);
4170914                           *ptr_size = value_i;
4170915                           assigned++;
4170916                         }
4170917                       f += 2;
4170918                       input = next;
4170919                     }
4170920                   else if (format[f + 1] == 'u')
4170921                     {
4170922                       // ----------------------------
4170923                       // size_t, base 10.
4170924                       // ----------------------------
4170925                       value_u =
4170926                         strtointmax (input, &next, 10,
4170927                                      width);
4170928                       if (input == next)
4170929                         {
4170930                           return (ass_or_eof
4170931                                   (consumed, assigned));
4170932                         }
4170933                       consumed++;
4170934                       if (!flag_star)
4170935                         {
4170936                           ptr_size = va_arg (ap, size_t *);
4170937                           *ptr_size = value_u;
4170938                           assigned++;
4170939                         }
4170940                       f += 2;
4170941                       input = next;
4170942                     }
4170943                   else if (format[f + 1] == 'x'
4170944                            || format[f + 2] == 'X')
4170945                     {
4170946                       // ----------------------------
4170947                       // size_t, base 16.
4170948                       // ----------------------------
4170949                       value_i =
4170950                         strtointmax (input, &next, 16,
4170951                                      width);
4170952                       if (input == next)
4170953                         {
4170954                           return (ass_or_eof
4170955                                   (consumed, assigned));
4170956                         }
4170957                       consumed++;
4170958                       if (!flag_star)
4170959                         {
```



```
4170960                           ptr_size = va_arg (ap, size_t *);
4170961                           *ptr_size = value_i;
4170962                           assigned++;
4170963                         }
4170964                       f += 2;
4170965                       input = next;
4170966                     }
4170967                   else if (format[f + 1] == 'n')
4170968                     {
4170969                       // ----------------------------
4170970                       // signed char,
4170971                       // string index counter.
4170972                       // ----------------------------
4170973                       ptr_size = va_arg (ap, size_t *);
4170974                       *ptr_size =
4170975                         (size_t) (input - start + scanned);
4170976                       f += 2;
4170977                     }
4170978                   else
4170979                     {
4170980                       // ----------------------------
4170981                       // unsupported or
4170982                       // unknown specifier.
4170983                       // ----------------------------
4170984                       f += 1;
4170985                     }
4170986                 }
4170987               else if (format[f] == 't')
4170988                 {
4170989                   // ---------------------- ptrdiff_t.
4170990                   if (format[f + 1] == 'd')
4170991                     {
4170992                       // ----------------------------
4170993                       // ptrdiff_t, base 10.
4170994                       // ----------------------------
4170995                       value_i =
4170996                         strtointmax (input, &next, 10,
4170997                                      width);
4170998                       if (input == next)
4170999                         {
4171000                           return (ass_or_eof
4171001                                   (consumed, assigned));
4171002                         }
4171003                       consumed++;
4171004                       if (!flag_star)
4171005                         {
4171006                           ptr_ptrdiff =
4171007                             va_arg (ap, ptrdiff_t *);
4171008                           *ptr_ptrdiff = value_i;
4171009                           assigned++;
4171010                         }
4171011                       f += 2;
4171012                       input = next;
4171013                     }
4171014                   else if (format[f + 1] == 'i')
4171015                     {
4171016                       // ----------------------------
4171017                       // ptrdiff_t, base unknown.
4171018                       // ----------------------------
4171019                       value_i =
4171020                         strtointmax (input, &next, 0,
4171021                                      width);
4171022                       if (input == next)
4171023                         {
4171024                           return (ass_or_eof
4171025                                   (consumed, assigned));
4171026                         }
4171027                       consumed++;
4171028                       if (!flag_star)
4171029                         {
4171030                           ptr_ptrdiff =
4171031                             va_arg (ap, ptrdiff_t *);
4171032                           *ptr_ptrdiff = value_i;
4171033                           assigned++;
4171034                         }
4171035                       f += 2;
4171036                       input = next;
4171037                     }
4171038                   else if (format[f + 1] == 'o')
4171039                     {
4171040                       // ----------------------------
4171041                       // ptrdiff_t, base 8.
4171042                       // ----------------------------
4171043                       value_i =
4171044                         strtointmax (input, &next, 8,
4171045                                      width);
4171046                       if (input == next)
```

```
                        {
                          return (ass_or_eof
                                  (consumed, assigned));
                        }
                      consumed++;
                      if (!flag_star)
                        {
                          ptr_ptrdiff =
                            va_arg (ap, ptrdiff_t *);
                          *ptr_ptrdiff = value_i;
                          assigned++;
                        }
                      f += 2;
                      input = next;
                    }
                  else if (format[f + 1] == 'u')
                    {
                      // ------------------------------
                      // ptrdiff_t, base 10.
                      // ------------------------------
                      value_u =
                        strtointmax (input, &next, 10,
                                     width);
                      if (input == next)
                        {
                          return (ass_or_eof
                                  (consumed, assigned));
                        }
                      consumed++;
                      if (!flag_star)
                        {
                          ptr_ptrdiff =
                            va_arg (ap, ptrdiff_t *);
                          *ptr_ptrdiff = value_u;
                          assigned++;
                        }
                      f += 2;
                      input = next;
                    }
                  else if (format[f + 1] == 'x'
                           || format[f + 2] == 'X')
                    {
                      // ------------------------------
                      // ptrdiff_t, base 16.
                      // ------------------------------
                      value_i =
                        strtointmax (input, &next, 16,
                                     width);
                      if (input == next)
                        {
                          return (ass_or_eof
                                  (consumed, assigned));
                        }
                      consumed++;
                      if (!flag_star)
                        {
                          ptr_ptrdiff =
                            va_arg (ap, ptrdiff_t *);
                          *ptr_ptrdiff = value_i;
                          assigned++;
                        }
                      f += 2;
                      input = next;
                    }
                  else if (format[f + 1] == 'n')
                    {
                      // ------------------------------
                      // signed char,
                      // string index counter.
                      // ------------------------------
                      ptr_ptrdiff =
                        va_arg (ap, ptrdiff_t *);
                      *ptr_ptrdiff =
                        (ptrdiff_t) (input - start +
                                     scanned);
                      f += 2;
                    }
                  else
                    {
                      // ------------------------------
                      // unsupported or
                      // unknown specifier.
                      // ------------------------------
                      f += 1;
                    }
                }
              //
```



```
              // Specifiers with no length modifier.
              //
              if (format[f] == 'd')
                {
                  // ---------- signed short, base 10.
                  value_i =
                    strtointmax (input, &next, 10, width);
                  if (input == next)
                    {
                      return (ass_or_eof
                              (consumed, assigned));
                    }
                  consumed++;
                  if (!flag_star)
                    {
                      ptr_sshort =
                        va_arg (ap, signed short *);
                      *ptr_sshort = value_i;
                      assigned++;
                    }
                  f += 1;
                  input = next;
                }
              else if (format[f] == 'i')
                {
                  // ---------------------------------
                  // signed
                  // int, base unknown.
                  // ---------------------------------
                  value_i =
                    strtointmax (input, &next, 0, width);
                  if (input == next)
                    {
                      return (ass_or_eof
                              (consumed, assigned));
                    }
                  consumed++;
                  if (!flag_star)
                    {
                      ptr_sint = va_arg (ap, signed int *);
                      *ptr_sint = value_i;
                      assigned++;
                    }
                  f += 1;
                  input = next;
                }
              else if (format[f] == 'o')
                {
                  // ---------------------------------
                  // signed int, base 8.
                  // ---------------------------------
                  value_i =
                    strtointmax (input, &next, 8, width);
                  if (input == next)
                    {
                      return (ass_or_eof
                              (consumed, assigned));
                    }
                  consumed++;
                  if (!flag_star)
                    {
                      ptr_sint = va_arg (ap, signed int *);
                      *ptr_sint = value_i;
                      assigned++;
                    }
                  f += 1;
                  input = next;
                }
              else if (format[f] == 'u')
                {
                  // ---------------------------------
                  // unsigned short, base 10.
                  // ---------------------------------
                  value_u =
                    strtointmax (input, &next, 10, width);
                  if (input == next)
                    {
                      return (ass_or_eof
                              (consumed, assigned));
                    }
                  consumed++;
                  if (!flag_star)
                    {
                      ptr_uint =
                        va_arg (ap, unsigned int *);
                      *ptr_uint = value_u;
                      assigned++;
```

```
                    }
                  f += 1;
                  input = next;
                }
              else if (format[f] == 'x' || format[f] == 'X')
                {
                  // ---------------------------------
                  // signed short, base 16.
                  // ---------------------------------
                  value_i =
                    strtointmax (input, &next, 16, width);
                  if (input == next)
                    {
                      return (ass_or_eof
                              (consumed, assigned));
                    }
                  consumed++;
                  if (!flag_star)
                    {
                      ptr_sint = va_arg (ap, signed int *);
                      *ptr_sint = value_i;
                      assigned++;
                    }
                  f += 1;
                  input = next;
                }
              else if (format[f] == 'c')
                {
                  // ------------------------- char[].
                  if (width == 0)
                    width = 1;
                  //
                  if (!flag_star)
                    ptr_char = va_arg (ap, char *);
                  //
                  for (count = 0;
                       width > 0 && *input != 0;
                       width--, ptr_char++, input++)
                    {
                      if (!flag_star)
                        *ptr_char = *input;
                      //
                      count++;
                    }
                  //
                  if (count)
                    consumed++;
                  if (count && !flag_star)
                    assigned++;
                  //
                  f += 1;
                }
              else if (format[f] == 's')
                {
                  // ------------------------- string.
                  if (!flag_star)
                    ptr_char = va_arg (ap, char *);
                  //
                  for (count = 0;
                       !isspace (*input)
                       && *input != 0; ptr_char++, input++)
                    {
                      if (!flag_star)
                        *ptr_char = *input;
                      //
                      count++;
                    }
                  if (!flag_star)
                    *ptr_char = 0;
                  //
                  if (count)
                    consumed++;
                  if (count && !flag_star)
                    assigned++;
                  //
                  f += 1;
                }
              else if (format[f] == '[')
                {
                  //
                  f++;
                  //
                  if (format[f] == '^')
                    {
                      inverted = 1;
                      f++;
                    }
```



```
                  else
                    {
                      inverted = 0;
                    }
                  //
                  // Reset ascii array.
                  //
                  for (index = 0; index < 128; index++)
                    {
                      ascii[index] = inverted;
                    }
                  //
                  //
                  //
                  for (count = 0;
                       &format[f] < end_format; count++)
                    {
                      if (format[f] == ']' && count > 0)
                        {
                          break;
                        }
                      //
                      // Check for an interval.
                      //
                      if (format[f + 1] == '-'
                          && format[f + 2] != ']'
                          && format[f + 2] != 0)
                        {
                          //
                          // Interval.
                          //
                          for (index = format[f];
                               index <= format[f + 2];
                               index++)
                            {
                              ascii[index] = !inverted;
                            }
                          f += 3;
                          continue;
                        }
                      //
                      // Single character.
                      //
                      index = format[f];
                      ascii[index] = !inverted;
                      f++;
                    }
                  //
                  // Is the scan correctly finished?.
                  //
                  if (format[f] != ']')
                    {
                      return (ass_or_eof
                              (consumed, assigned));
                    }
                  //
                  // The ascii table is populated.
                  //
                  if (width == 0)
                    width = SIZE_MAX;
                  //
                  // Scan the input string.
                  //
                  if (!flag_star)
                    ptr_char = va_arg (ap, char *);
                  //
                  for (count = 0;
                       width > 0 && *input != 0;
                       width--, ptr_char++, input++)
                    {
                      index = *input;
                      if (ascii[index])
                        {
                          if (!flag_star)
                            *ptr_char = *input;
                          count++;
                        }
                      else
                        {
                          break;
                        }
                    }
                  //
                  if (count)
                    consumed++;
                  if (count && !flag_star)
                    assigned++;
```

```
                    //
                    f += 1;
                  }
                else if (format[f] == 'p')
                  {
                    // ------------------------ void *.
                    value_i =
                      strtointmax (input, &next, 16, width);
                    if (input == next)
                      {
                        return (ass_or_eof
                                (consumed, assigned));
                      }
                    consumed++;
                    if (!flag_star)
                      {
                        ptr_void = va_arg (ap, void **);
                        *ptr_void = (void *) ((int) value_i);
                        assigned++;
                      }
                    f += 1;
                    input = next;
                  }
                else if (format[f] == 'n')
                  {
                    // ---------------------------------
                    // signed char,
                    // string index counter.
                    // ---------------------------------
                    ptr_sint = va_arg (ap, signed int *);
                    *ptr_sint =
                      (signed char) (input - start + scanned);
                    f += 1;
                  }
                else
                  {
                    // ---------------------------------
                    // unsupported or
                    // unknown specifier.
                    // ---------------------------------
                    ;
                  }

                // -------------------------------------
                // End of specifier.
                // -------------------------------------

                width_string[0] = '\0';
                specifier = 0;
                specifier_flags = 0;
                specifier_width = 0;
                specifier_type = 0;
                flag_star = 0;

              }
          }
        //
        // The format or the input string is terminated.
        //
        if (&format[f] < end_format && stream)
          {
            //
            // Only the input string is finished, and
            // the input comes
            // from a stream, so another read will be
            // done.
            //
            scanned += (int) (input - start);
            continue;
          }
        //
        // The format string is terminated.
        //
        return (ass_or_eof (consumed, assigned));
      }
}

//-----------------------------------------------------------
static intmax_t
strtointmax (const char *restrict string,
             const char **restrict endptr, int base,
             size_t max_width)
{
  int i;
  int d;        // Digits counter.
  int sign = +1;
  intmax_t number;
```



```
  intmax_t previous;
  int digit;
  //
  bool flag_prefix_oct = 0;
  bool flag_prefix_exa = 0;
  bool flag_prefix_dec = 0;
  //
  // If the 'max_width' value is zero, fix it to the
  // maximum
  // that it can represent.
  //
  if (max_width == 0)
    {
      max_width = SIZE_MAX;
    }
  //
  // Eat initial spaces, but if there are spaces,
  // there is an
  // error inside the calling function!
  //
  for (i = 0; isspace (string[i]); i++)
    {
      fprintf (stderr,
               "libc error: file \"%s\", line %i\n",
               __FILE__, __LINE__);
      ;
    }
  //
  // Check sign. The 'max_width' counts also the sign,
  // if there is
  // one.
  //
  if (string[i] == '+')
    {
      sign = +1;
      i++;
      max_width--;
    }
  else if (string[i] == '-')
    {
      sign = -1;
      i++;
      max_width--;
    }
  //
  // Check for prefix.
  //
  if (string[i] == '0')
    {
      if (string[i + 1] == 'x' || string[i + 1] == 'X')
        {
          flag_prefix_exa = 1;
        }
      if (isdigit (string[i + 1]))
        {
          flag_prefix_oct = 1;
        }
    }
  //
  if (string[i] > '0' && string[i] <= '9')
    {
      flag_prefix_dec = 1;
    }
  //
  // Check compatibility with requested base.
  //
  if (flag_prefix_exa)
    {
      if (base == 0)
        {
          base = 16;
        }
      else if (base == 16)
        {
          ;     // Ok.
        }
      else
        {
          //
          // Incompatible sequence: only the initial
          // zero is reported.
          //
          *endptr = &string[i + 1];
          return ((intmax_t) 0);
        }
      //
      // Move on, after the '0x' prefix.
```

```
4171569       //
4171570       i += 2;
4171571     }
4171572   //
4171573   if (flag_prefix_oct)
4171574     {
4171575       if (base == 0)
4171576         {
4171577           base = 8;
4171578         }
4171579       //
4171580       // Move on, after the '0' prefix.
4171581       //
4171582       i += 1;
4171583     }
4171584   //
4171585   if (flag_prefix_dec)
4171586     {
4171587       if (base == 0)
4171588         {
4171589           base = 10;
4171590         }
4171591     }
4171592   //
4171593   // Scan the string.
4171594   //
4171595   for (d = 0, number = 0;
4171596        d < max_width && string[i] != 0; i++, d++)
4171597     {
4171598       if (string[i] >= '0' && string[i] <= '9')
4171599         {
4171600           digit = string[i] - '0';
4171601         }
4171602       else if (string[i] >= 'A' && string[i] <= 'F')
4171603         {
4171604           digit = string[i] - 'A' + 10;
4171605         }
4171606       else if (string[i] >= 'a' && string[i] <= 'f')
4171607         {
4171608           digit = string[i] - 'a' + 10;
4171609         }
4171610       else
4171611         {
4171612           digit = 999;
4171613         }
4171614       //
4171615       // Give a sign to the digit.
4171616       //
4171617       digit *= sign;
4171618       //
4171619       // Compare with the base.
4171620       //
4171621       if (base > (digit * sign))
4171622         {
4171623           //
4171624           // Check if the current digit can be safely
4171625           // computed.
4171626           //
4171627           previous = number;
4171628           number *= base;
4171629           number += digit;
4171630           if (number / base != previous)
4171631             {
4171632               //
4171633               // Out of range.
4171634               //
4171635               *endptr = &string[i + 1];
4171636               errset (ERANGE);  // Result too large.
4171637               if (sign > 0)
4171638                 {
4171639                   return (INTMAX_MAX);
4171640                 }
4171641               else
4171642                 {
4171643                   return (INTMAX_MIN);
4171644                 }
4171645             }
4171646         }
4171647       else
4171648         {
4171649           *endptr = &string[i];
4171650           return (number);
4171651         }
4171652     }
4171653   //
4171654   // The string is finished or the max digits length
4171655   // is reached.
4171656   //
4171657   *endptr = &string[i];
4171658   //
4171659   return (number);
4171660 }
4171661
4171662 //-----------------------------------------------------------
4171663 static int
4171664 ass_or_eof (int consumed, int assigned)
4171665 {
4171666   if (consumed == 0)
4171667     {
4171668       return (EOF);
4171669     }
4171670   else
4171671     {
4171672       return (assigned);
4171673     }
4171674 }
4171675
4171676 //-----------------------------------------------------------
```


95.18.40 lib/stdio/vprintf.c

Si veda la sezione 88.137.

```
4180001 #include <stdio.h>
4180002 #include <sys/types.h>
4180003 #include <sys/os32.h>
4180004 #include <string.h>
4180005 #include <unistd.h>
4180006 //-----------------------------------------------------------
4180007 int
4180008 vprintf (const char *restrict format, va_list arg)
4180009 {
4180010   ssize_t size_written;
4180011   size_t size;
4180012   size_t size_total;
4180013   int status;
4180014   char string[BUFSIZ];
4180015   char *buffer = string;
4180016
4180017   buffer[0] = 0;
4180018   status = vsprintf (buffer, format, arg);
4180019
4180020   size = strlen (buffer);
4180021   if (size >= BUFSIZ)
4180022     {
4180023       size = BUFSIZ;
4180024     }
4180025
4180026   for (size_total = 0, size_written = 0;
4180027        size_total < size;
4180028        size_total += size_written, buffer += size_written)
4180029     {
4180030       //
4180031       // Write to the standard output: file descriptor
4180032       // n. 1.
4180033       //
4180034       size_written =
4180035         write (STDOUT_FILENO, buffer, size - size_total);
4180036       if (size_written < 0)
4180037         {
4180038           return (size_total);
4180039         }
4180040     }
4180041   return (size);
4180042 }
```

95.18.41 lib/stdio/vscanf.c

Si veda la sezione 88.138.

```
4190001 #include <stdio.h>
4190002 //-----------------------------------------------------------
4190003 int
4190004 vscanf (const char *restrict format, va_list ap)
4190005 {
4190006   return (vfscanf (stdin, format, ap));
4190007 }
4190008
4190009 //-----------------------------------------------------------
```

95.18.42 lib/stdio/vsnprintf.c

«

Si veda la sezione 88.137.

```
4200001  #include <stdint.h>
4200002  #include <stdbool.h>
4200003  #include <stdlib.h>
4200004  #include <string.h>
4200005  #include <stdio.h>
4200006  //-------------------------------------------------------
4200007  static size_t uimaxtoa (uintmax_t integer,
4200008                          char *buffer, int base,
4200009                          int uppercase, size_t size);
4200010  static size_t imaxtoa (intmax_t integer, char *buffer,
4200011                         int base, int uppercase,
4200012                         size_t size);
4200013  static size_t simaxtoa (intmax_t integer, char *buffer,
4200014                          int base, int uppercase,
4200015                          size_t size);
4200016  static size_t uimaxtoa_fill (uintmax_t integer,
4200017                               char *buffer, int base,
4200018                               int uppercase, int width,
4200019                               int filler, int max);
4200020  static size_t imaxtoa_fill (intmax_t integer,
4200021                              char *buffer, int base,
4200022                              int uppercase, int width,
4200023                              int filler, int max);
4200024  static size_t simaxtoa_fill (intmax_t integer,
4200025                               char *buffer, int base,
4200026                               int uppercase, int width,
4200027                               int filler, int max);
4200028  static size_t strtostr_fill (char *string,
4200029                               char *buffer, int width,
4200030                               int filler, int max);
4200031  //-------------------------------------------------------
4200032  int
4200033  vsnprintf (char *restrict string, size_t size,
4200034             const char *restrict format, va_list ap)
4200035  {
4200036    //
4200037    // We produce at most 'size-1' characters, + '\0'.
4200038    // 'size' is used also as the max size for internal
4200039    // strings, but only if it is not too big.
4200040    //
4200041    int f = 0;
4200042    int s = 0;
4200043    int remain = size - 1;
4200044    //
4200045    bool specifier = 0;
4200046    bool specifier_flags = 0;
4200047    bool specifier_width = 0;
4200048    bool specifier_precision = 0;
4200049    bool specifier_type = 0;
4200050    //
4200051    bool flag_plus = 0;
4200052    bool flag_minus = 0;
4200053    bool flag_space = 0;
4200054    bool flag_alternate = 0;
4200055    bool flag_zero = 0;
4200056    //
4200057    int alignment;
4200058    int filler;
4200059    //
4200060    intmax_t value_i;
4200061    uintmax_t value_ui;
4200062    char *value_cp;
4200063    //
4200064    size_t width;
4200065    size_t precision;
4200066    size_t str_size =
4200067      (size > (BUFSIZ / 2) ? (BUFSIZ / 2) : size);
4200068    char width_string[str_size];
4200069    char precision_string[str_size];
4200070    int w;
4200071    int p;
4200072    //
4200073    width_string[0] = '\0';
4200074    precision_string[0] = '\0';
4200075    //
4200076    while (format[f] != 0 && s < (size - 1))
4200077      {
4200078        if (!specifier)
4200079          {
4200080            // ------------------- The context is not
4200081            // inside a specifier.
4200082            if (format[f] != '%')
4200083              {
4200084                string[s] = format[f];
4200085                s++;
4200086                remain--;
4200087                f++;
4200088                continue;
4200089              }
4200090            if (format[f] == '%' && format[f + 1] == '%')
4200091              {
4200092                string[s] = '%';
4200093                f++;
4200094                f++;
4200095                s++;
4200096                remain--;
4200097                continue;
4200098              }
4200099            if (format[f] == '%')
4200100              {
4200101                f++;
4200102                specifier = 1;
4200103                specifier_flags = 1;
4200104                continue;
4200105              }
4200106          }
4200107        //
4200108        if (specifier && specifier_flags)
4200109          {
4200110            // ------------------- The context is inside
4200111            // specifier flags.
4200112            if (format[f] == '+')
4200113              {
4200114                flag_plus = 1;
4200115                f++;
4200116                continue;
4200117              }
4200118            else if (format[f] == '-')
4200119              {
4200120                flag_minus = 1;
4200121                f++;
4200122                continue;
4200123              }
4200124            else if (format[f] == ' ')
4200125              {
4200126                flag_space = 1;
4200127                f++;
4200128                continue;
4200129              }
4200130            else if (format[f] == '#')
4200131              {
4200132                flag_alternate = 1;
4200133                f++;
4200134                continue;
4200135              }
4200136            else if (format[f] == '0')
4200137              {
4200138                flag_zero = 1;
4200139                f++;
4200140                continue;
4200141              }
4200142            else
4200143              {
4200144                specifier_flags = 0;
4200145                specifier_width = 1;
4200146              }
4200147          }
4200148        //
4200149        if (specifier && specifier_width)
4200150          {
4200151            // ------------------- The context is inside
4200152            // specifier width.
4200153            for (w = 0;
4200154                 format[f] >= '0' && format[f] <= '9'
4200155                 && w < str_size; w++)
4200156              {
4200157                width_string[w] = format[f];
4200158                f++;
4200159              }
4200160            width_string[w] = '\0';
4200161
4200162            specifier_width = 0;
4200163
4200164            if (format[f] == '.')
4200165              {
4200166                specifier_precision = 1;
4200167                f++;
4200168              }
4200169            else
4200170              {
4200171                specifier_precision = 0;
4200172                specifier_type = 1;
```

```
4200173             }
4200174           }
4200175         //
4200176         if (specifier && specifier_precision)
4200177           {
4200178             // --------------- The context is inside
4200179             // specifier precision.
4200180             for (p = 0;
4200181                  format[f] >= '0' && format[f] <= '9'
4200182                  && p < str_size; p++)
4200183               {
4200184                 precision_string[p] = format[f];
4200185                 p++;
4200186               }
4200187             precision_string[p] = '\0';
4200188
4200189             specifier_precision = 0;
4200190             specifier_type = 1;
4200191           }
4200192         //
4200193         if (specifier && specifier_type)
4200194           {
4200195             // -------------------- The context is
4200196             // inside specifier type.
4200197             width = atoi (width_string);
4200198             precision = atoi (precision_string);
4200199             filler = ' ';
4200200             if (flag_zero)
4200201               filler = '0';
4200202             if (flag_space)
4200203               filler = ' ';
4200204             alignment = width;
4200205             if (flag_minus)
4200206               {
4200207                 alignment = -alignment;
4200208                 filler = ' ';     // The filler
4200209                 // character cannot
4200210                 // be zero, so it is black.
4200211               }
4200212             //
4200213             if (format[f] == 'h' && format[f + 1] == 'h')
4200214               {
4200215                 if (format[f + 2] == 'd'
4200216                     || format[f + 2] == 'i')
4200217                   {
4200218                     // ----------------------------
4200219                     // signed char, base 10.
4200220                     value_i = va_arg (ap, int);
4200221                     if (flag_plus)
4200222                       {
4200223                         s +=
4200224                           simaxtoa_fill (value_i,
4200225                                          &string[s], 10,
4200226                                          0, alignment,
4200227                                          filler, remain);
4200228                       }
4200229                     else
4200230                       {
4200231                         s +=
4200232                           imaxtoa_fill (value_i,
4200233                                         &string[s], 10,
4200234                                         0, alignment,
4200235                                         filler, remain);
4200236                       }
4200237                     f += 3;
4200238                   }
4200239                 else if (format[f + 2] == 'u')
4200240                   {
4200241                     // -------------------------
4200242                     // unsigned char, base 10.
4200243                     value_ui = va_arg (ap, unsigned int);
4200244                     s +=
4200245                       uimaxtoa_fill (value_ui,
4200246                                      &string[s], 10, 0,
4200247                                      alignment, filler,
4200248                                      remain);
4200249                     f += 3;
4200250                   }
4200251                 else if (format[f + 2] == 'o')
4200252                   {
4200253                     // -------------------------
4200254                     // unsigned char, base 8.
4200255                     value_ui = va_arg (ap, unsigned int);
4200256                     s +=
4200257                       uimaxtoa_fill (value_ui,
4200258                                      &string[s], 8, 0,
4200259                                      alignment, filler,
```

```
4200260                                      remain);
4200261                     f += 3;
4200262                   }
4200263                 else if (format[f + 2] == 'x')
4200264                   {
4200265                     // -------------------------
4200266                     // unsigned char, base 16.
4200267                     value_ui = va_arg (ap, unsigned int);
4200268                     s +=
4200269                       uimaxtoa_fill (value_ui,
4200270                                      &string[s], 16, 0,
4200271                                      alignment, filler,
4200272                                      remain);
4200273                     f += 3;
4200274                   }
4200275                 else if (format[f + 2] == 'X')
4200276                   {
4200277                     // -------------------------
4200278                     // unsigned char, base 16.
4200279                     value_ui = va_arg (ap, unsigned int);
4200280                     s +=
4200281                       uimaxtoa_fill (value_ui,
4200282                                      &string[s], 16, 1,
4200283                                      alignment, filler,
4200284                                      remain);
4200285                     f += 3;
4200286                   }
4200287                 else if (format[f + 2] == 'b')
4200288                   {
4200289                     // --------------- unsigned char,
4200290                     // base 2 (extention).
4200291                     value_ui = va_arg (ap, unsigned int);
4200292                     s +=
4200293                       uimaxtoa_fill (value_ui,
4200294                                      &string[s], 2, 0,
4200295                                      alignment, filler,
4200296                                      remain);
4200297                     f += 3;
4200298                   }
4200299                 else
4200300                   {
4200301                     // ---------------- unsupported or
4200302                     // unknown specifier.
4200303                     f += 2;
4200304                   }
4200305               }
4200306             else if (format[f] == 'h')
4200307               {
4200308                 if (format[f + 1] == 'd'
4200309                     || format[f + 1] == 'i')
4200310                   {
4200311                     // -----------------------------
4200312                     // short int, base 10.
4200313                     value_i = va_arg (ap, int);
4200314                     if (flag_plus)
4200315                       {
4200316                         s +=
4200317                           simaxtoa_fill (value_i,
4200318                                          &string[s], 10,
4200319                                          0, alignment,
4200320                                          filler, remain);
4200321                       }
4200322                     else
4200323                       {
4200324                         s +=
4200325                           imaxtoa_fill (value_i,
4200326                                         &string[s], 10,
4200327                                         0, alignment,
4200328                                         filler, remain);
4200329                       }
4200330                     f += 2;
4200331                   }
4200332                 else if (format[f + 1] == 'u')
4200333                   {
4200334                     // --------------------- unsigned
4200335                     // short int, base 10.
4200336                     value_ui = va_arg (ap, unsigned int);
4200337                     s +=
4200338                       uimaxtoa_fill (value_ui,
4200339                                      &string[s], 10, 0,
4200340                                      alignment, filler,
4200341                                      remain);
4200342                     f += 2;
4200343                   }
4200344                 else if (format[f + 1] == 'o')
4200345                   {
4200346                     // ---------------------- unsigned
```

```
4200347                  // short int, base 8.
4200348                  value_ui = va_arg (ap, unsigned int);
4200349                  s +=
4200350                    uimaxtoa_fill (value_ui,
4200351                                   &string[s], 8, 0,
4200352                                   alignment, filler,
4200353                                   remain);
4200354                  f += 2;
4200355                }
4200356              else if (format[f + 1] == 'x')
4200357                {
4200358                  // -------------------- unsigned
4200359                  // short int, base 16.
4200360                  value_ui = va_arg (ap, unsigned int);
4200361                  s +=
4200362                    uimaxtoa_fill (value_ui,
4200363                                   &string[s], 16, 0,
4200364                                   alignment, filler,
4200365                                   remain);
4200366                  f += 2;
4200367                }
4200368              else if (format[f + 1] == 'X')
4200369                {
4200370                  // -------------------- unsigned
4200371                  // short int, base 16.
4200372                  value_ui = va_arg (ap, unsigned int);
4200373                  s +=
4200374                    uimaxtoa_fill (value_ui,
4200375                                   &string[s], 16, 1,
4200376                                   alignment, filler,
4200377                                   remain);
4200378                  f += 2;
4200379                }
4200380              else if (format[f + 1] == 'b')
4200381                {
4200382                  // ---------- unsigned short int,
4200383                  // base 2 (extention).
4200384                  value_ui = va_arg (ap, unsigned int);
4200385                  s +=
4200386                    uimaxtoa_fill (value_ui,
4200387                                   &string[s], 2, 0,
4200388                                   alignment, filler,
4200389                                   remain);
4200390                  f += 2;
4200391                }
4200392              else
4200393                {
4200394                  // --------------- unsupported or
4200395                  // unknown specifier.
4200396                  f += 1;
4200397                }
4200398            }
4200399          else if (format[f] == 'l' && format[f + 1] != 'l')
4200400            {
4200401              if (format[f + 1] == 'd'
4200402                  || format[f + 1] == 'i')
4200403                {
4200404                  // -------------------------------
4200405                  // long int base 10.
4200406                  value_i = va_arg (ap, long int);
4200407                  if (flag_plus)
4200408                    {
4200409                      s +=
4200410                        simaxtoa_fill (value_i,
4200411                                       &string[s], 10,
4200412                                       0, alignment,
4200413                                       filler, remain);
4200414                    }
4200415                  else
4200416                    {
4200417                      s +=
4200418                        imaxtoa_fill (value_i,
4200419                                      &string[s], 10,
4200420                                      0, alignment,
4200421                                      filler, remain);
4200422                    }
4200423                  f += 2;
4200424                }
4200425              else if (format[f + 1] == 'u')
4200426                {
4200427                  // ---------------------- Unsigned
4200428                  // long int base 10.
4200429                  value_ui = va_arg (ap, unsigned long int);
4200430                  s +=
4200431                    uimaxtoa_fill (value_ui,
4200432                                   &string[s], 10, 0,
4200433                                   alignment, filler,
```

```
4200434                                   remain);
4200435                  f += 2;
4200436                }
4200437              else if (format[f + 1] == 'o')
4200438                {
4200439                  // ---------------------- Unsigned
4200440                  // long int base 8.
4200441                  value_ui = va_arg (ap, unsigned long int);
4200442                  s +=
4200443                    uimaxtoa_fill (value_ui,
4200444                                   &string[s], 8, 0,
4200445                                   alignment, filler,
4200446                                   remain);
4200447                  f += 2;
4200448                }
4200449              else if (format[f + 1] == 'x')
4200450                {
4200451                  // ---------------------- Unsigned
4200452                  // long int base 16.
4200453                  value_ui = va_arg (ap, unsigned long int);
4200454                  s +=
4200455                    uimaxtoa_fill (value_ui,
4200456                                   &string[s], 16, 0,
4200457                                   alignment, filler,
4200458                                   remain);
4200459                  f += 2;
4200460                }
4200461              else if (format[f + 1] == 'X')
4200462                {
4200463                  // ---------------------- Unsigned
4200464                  // long int base 16.
4200465                  value_ui = va_arg (ap, unsigned long int);
4200466                  s +=
4200467                    uimaxtoa_fill (value_ui,
4200468                                   &string[s], 16, 1,
4200469                                   alignment, filler,
4200470                                   remain);
4200471                  f += 2;
4200472                }
4200473              else if (format[f + 1] == 'b')
4200474                {
4200475                  // ------------ Unsigned long int
4200476                  // base 2 (extention).
4200477                  value_ui = va_arg (ap, unsigned long int);
4200478                  s +=
4200479                    uimaxtoa_fill (value_ui,
4200480                                   &string[s], 2, 0,
4200481                                   alignment, filler,
4200482                                   remain);
4200483                  f += 2;
4200484                }
4200485              else
4200486                {
4200487                  // --------------- unsupported or
4200488                  // unknown specifier.
4200489                  f += 1;
4200490                }
4200491            }
4200492          else if (format[f] == 'l' && format[f + 1] == 'l')
4200493            {
4200494              if (format[f + 2] == 'd'
4200495                  || format[f + 2] == 'i')
4200496                {
4200497                  // -------------------------------
4200498                  // long int base 10.
4200499                  value_i = va_arg (ap, long long int);
4200500                  if (flag_plus)
4200501                    {
4200502                      s +=
4200503                        simaxtoa_fill (value_i,
4200504                                       &string[s], 10,
4200505                                       0, alignment,
4200506                                       filler, remain);
4200507                    }
4200508                  else
4200509                    {
4200510                      s +=
4200511                        imaxtoa_fill (value_i,
4200512                                      &string[s], 10,
4200513                                      0, alignment,
4200514                                      filler, remain);
4200515                    }
4200516                  f += 3;
4200517                }
4200518              else if (format[f + 2] == 'u')
4200519                {
4200520                  // ---------------------- Unsigned
```

```
4200521                 // long int base 10.
4200522                 value_ui =
4200523                   va_arg (ap, unsigned long long int);
4200524                 s +=
4200525                   uimaxtoa_fill (value_ui,
4200526                                  &string[s], 10, 0,
4200527                                  alignment, filler,
4200528                                  remain);
4200529                 f += 3;
4200530               }
4200531             else if (format[f + 2] == 'o')
4200532               {
4200533                 // ----------------------- Unsigned
4200534                 // long int base 8.
4200535                 value_ui =
4200536                   va_arg (ap, unsigned long long int);
4200537                 s +=
4200538                   uimaxtoa_fill (value_ui,
4200539                                  &string[s], 8, 0,
4200540                                  alignment, filler,
4200541                                  remain);
4200542                 f += 3;
4200543               }
4200544             else if (format[f + 2] == 'x')
4200545               {
4200546                 // ----------------------- Unsigned
4200547                 // long int base 16.
4200548                 value_ui =
4200549                   va_arg (ap, unsigned long long int);
4200550                 s +=
4200551                   uimaxtoa_fill (value_ui,
4200552                                  &string[s], 16, 0,
4200553                                  alignment, filler,
4200554                                  remain);
4200555                 f += 3;
4200556               }
4200557             else if (format[f + 2] == 'X')
4200558               {
4200559                 // ----------------------- Unsigned
4200560                 // long int base 16.
4200561                 value_ui =
4200562                   va_arg (ap, unsigned long long int);
4200563                 s +=
4200564                   uimaxtoa_fill (value_ui,
4200565                                  &string[s], 16, 1,
4200566                                  alignment, filler,
4200567                                  remain);
4200568                 f += 3;
4200569               }
4200570             else if (format[f + 2] == 'b')
4200571               {
4200572                 // ------------ Unsigned long int
4200573                 // base 2 (extention).
4200574                 value_ui =
4200575                   va_arg (ap, unsigned long long int);
4200576                 s +=
4200577                   uimaxtoa_fill (value_ui,
4200578                                  &string[s], 2, 0,
4200579                                  alignment, filler,
4200580                                  remain);
4200581                 f += 3;
4200582               }
4200583             else
4200584               {
4200585                 // ---------------- unsupported or
4200586                 // unknown specifier.
4200587                 f += 2;
4200588               }
4200589           }
4200590         else if (format[f] == 'j')
4200591           {
4200592             if (format[f + 1] == 'd'
4200593                 || format[f + 1] == 'i')
4200594               {
4200595                 // --------------------------------
4200596                 // intmax_t base 10.
4200597                 value_i = va_arg (ap, intmax_t);
4200598                 if (flag_plus)
4200599                   {
4200600                     s +=
4200601                       simaxtoa_fill (value_i,
4200602                                      &string[s], 10,
4200603                                      0, alignment,
4200604                                      filler, remain);
4200605                   }
4200606                 else
4200607                   {
```



```
4200608                     s +=
4200609                       imaxtoa_fill (value_i,
4200610                                     &string[s], 10,
4200611                                     0, alignment,
4200612                                     filler, remain);
4200613                   }
4200614                 f += 2;
4200615               }
4200616             else if (format[f + 1] == 'u')
4200617               {
4200618                 // --------------------------------
4200619                 // uintmax_t base 10.
4200620                 value_ui = va_arg (ap, uintmax_t);
4200621                 s +=
4200622                   uimaxtoa_fill (value_ui,
4200623                                  &string[s], 10, 0,
4200624                                  alignment, filler,
4200625                                  remain);
4200626                 f += 2;
4200627               }
4200628             else if (format[f + 1] == 'o')
4200629               {
4200630                 // --------------------------------
4200631                 // uintmax_t base 8.
4200632                 value_ui = va_arg (ap, uintmax_t);
4200633                 s +=
4200634                   uimaxtoa_fill (value_ui,
4200635                                  &string[s], 8, 0,
4200636                                  alignment, filler,
4200637                                  remain);
4200638                 f += 2;
4200639               }
4200640             else if (format[f + 1] == 'x')
4200641               {
4200642                 // --------------------------------
4200643                 // uintmax_t base 16.
4200644                 value_ui = va_arg (ap, uintmax_t);
4200645                 s +=
4200646                   uimaxtoa_fill (value_ui,
4200647                                  &string[s], 16, 0,
4200648                                  alignment, filler,
4200649                                  remain);
4200650                 f += 2;
4200651               }
4200652             else if (format[f + 1] == 'X')
4200653               {
4200654                 // --------------------------------
4200655                 // uintmax_t base 16.
4200656                 value_ui = va_arg (ap, uintmax_t);
4200657                 s +=
4200658                   uimaxtoa_fill (value_ui,
4200659                                  &string[s], 16, 1,
4200660                                  alignment, filler,
4200661                                  remain);
4200662                 f += 2;
4200663               }
4200664             else if (format[f + 1] == 'b')
4200665               {
4200666                 // -------------------- uintmax_t
4200667                 // base 2 (extention).
4200668                 value_ui = va_arg (ap, uintmax_t);
4200669                 s +=
4200670                   uimaxtoa_fill (value_ui,
4200671                                  &string[s], 2, 0,
4200672                                  alignment, filler,
4200673                                  remain);
4200674                 f += 2;
4200675               }
4200676             else
4200677               {
4200678                 // ---------------- unsupported or
4200679                 // unknown specifier.
4200680                 f += 1;
4200681               }
4200682           }
4200683         else if (format[f] == 'z')
4200684           {
4200685             if (format[f + 1] == 'd'
4200686                 || format[f + 1] == 'i'
4200687                 || format[f + 1] == 'i')
4200688               {
4200689                 // ----------------- size_t base 10.
4200690                 value_ui = va_arg (ap, unsigned long int);
4200691                 s +=
4200692                   uimaxtoa_fill (value_ui,
4200693                                  &string[s], 10, 0,
4200694                                  alignment, filler,
```

```
                                    remain);
                    f += 2;
                  }
                else if (format[f + 1] == 'o')
                  {
                    // ------------------ size_t base 8.
                    value_ui = va_arg (ap, unsigned long int);
                    s +=
                      uimaxtoa_fill (value_ui,
                                     &string[s], 8, 0,
                                     alignment, filler,
                                     remain);
                    f += 2;
                  }
                else if (format[f + 1] == 'x')
                  {
                    // ----------------- size_t base 16.
                    value_ui = va_arg (ap, unsigned long int);
                    s +=
                      uimaxtoa_fill (value_ui,
                                     &string[s], 16, 0,
                                     alignment, filler,
                                     remain);
                    f += 2;
                  }
                else if (format[f + 1] == 'X')
                  {
                    // ----------------- size_t base 16.
                    value_ui = va_arg (ap, unsigned long int);
                    s +=
                      uimaxtoa_fill (value_ui,
                                     &string[s], 16, 1,
                                     alignment, filler,
                                     remain);
                    f += 2;
                  }
                else if (format[f + 1] == 'b')
                  {
                    // ----------------------- size_t
                    // base 2 (extention).
                    value_ui = va_arg (ap, unsigned long int);
                    s +=
                      uimaxtoa_fill (value_ui,
                                     &string[s], 2, 0,
                                     alignment, filler,
                                     remain);
                    f += 2;
                  }
                else
                  {
                    // ---------------- unsupported or
                    // unknown specifier.
                    f += 1;
                  }
              }
            else if (format[f] == 't')
              {
                if (format[f + 1] == 'd'
                    || format[f + 1] == 'i')
                  {
                    // ------------------------------
                    // ptrdiff_t base 10.
                    value_i = va_arg (ap, long int);
                    if (flag_plus)
                      {
                        s +=
                          simaxtoa_fill (value_i,
                                         &string[s], 10,
                                         0, alignment,
                                         filler, remain);
                      }
                    else
                      {
                        s +=
                          imaxtoa_fill (value_i,
                                        &string[s], 10,
                                        0, alignment,
                                        filler, remain);
                      }
                    f += 2;
                  }
                else if (format[f + 1] == 'u')
                  {
                    // ----------------- ptrdiff_t base
                    // 10, without sign.
                    value_ui = va_arg (ap, unsigned long int);
                    s +=
                      uimaxtoa_fill (value_ui,
                                     &string[s], 10, 0,
                                     alignment, filler,
                                     remain);
                    f += 2;
                  }
                else if (format[f + 1] == 'o')
                  {
                    // ----------------- ptrdiff_t base
                    // 8, without sign.
                    value_ui = va_arg (ap, unsigned long int);
                    s +=
                      uimaxtoa_fill (value_ui,
                                     &string[s], 8, 0,
                                     alignment, filler,
                                     remain);
                    f += 2;
                  }
                else if (format[f + 1] == 'x')
                  {
                    // ----------------- ptrdiff_t base
                    // 16, without sign.
                    value_ui = va_arg (ap, unsigned long int);
                    s +=
                      uimaxtoa_fill (value_ui,
                                     &string[s], 16, 0,
                                     alignment, filler,
                                     remain);
                    f += 2;
                  }
                else if (format[f + 1] == 'X')
                  {
                    // ----------------- ptrdiff_t base
                    // 16, without sign.
                    value_ui = va_arg (ap, unsigned long int);
                    s +=
                      uimaxtoa_fill (value_ui,
                                     &string[s], 16, 1,
                                     alignment, filler,
                                     remain);
                    f += 2;
                  }
                else if (format[f + 1] == 'b')
                  {
                    // ------ ptrdiff_t base 2, without
                    // sign (extention).
                    value_ui = va_arg (ap, unsigned long int);
                    s +=
                      uimaxtoa_fill (value_ui,
                                     &string[s], 2, 0,
                                     alignment, filler,
                                     remain);
                    f += 2;
                  }
                else
                  {
                    // ---------------- unsupported or
                    // unknown specifier.
                    f += 1;
                  }
              }
            if (format[f] == 'd' || format[f] == 'i')
              {
                // ------------------------ int base 10.
                value_i = va_arg (ap, int);
                if (flag_plus)
                  {
                    s +=
                      simaxtoa_fill (value_i, &string[s],
                                     10, 0, alignment,
                                     filler, remain);
                  }
                else
                  {
                    s +=
                      imaxtoa_fill (value_i, &string[s],
                                    10, 0, alignment,
                                    filler, remain);
                  }
                f += 1;
              }
            else if (format[f] == 'u')
              {
                // ------------------------------
                // unsigned int base 10.
                value_ui = va_arg (ap, unsigned int);
                s +=
```

```
4200869                 uimaxtoa_fill (value_ui, &string[s],
4200870                                10, 0, alignment,
4200871                                filler, remain);
4200872               f += 1;
4200873             }
4200874           else if (format[f] == 'o')
4200875             {
4200876               // ---------------- unsigned int base 8.
4200877               value_ui = va_arg (ap, unsigned int);
4200878               s +=
4200879                 uimaxtoa_fill (value_ui, &string[s], 8,
4200880                                0, alignment, filler,
4200881                                remain);
4200882               f += 1;
4200883             }
4200884           else if (format[f] == 'x')
4200885             {
4200886               // -------------------------------
4200887               // unsigned int base 16.
4200888               value_ui = va_arg (ap, unsigned int);
4200889               s +=
4200890                 uimaxtoa_fill (value_ui, &string[s],
4200891                                16, 0, alignment,
4200892                                filler, remain);
4200893               f += 1;
4200894             }
4200895           else if (format[f] == 'X')
4200896             {
4200897               // -------------------------------
4200898               // unsigned int base 16.
4200899               value_ui = va_arg (ap, unsigned int);
4200900               s +=
4200901                 uimaxtoa_fill (value_ui, &string[s],
4200902                                16, 1, alignment,
4200903                                filler, remain);
4200904               f += 1;
4200905             }
4200906           else if (format[f] == 'b')
4200907             {
4200908               // --------------------- unsigned int
4200909               // base 2 (extention).
4200910               value_ui = va_arg (ap, unsigned int);
4200911               s +=
4200912                 uimaxtoa_fill (value_ui, &string[s], 2,
4200913                                0, alignment, filler,
4200914                                remain);
4200915               f += 1;
4200916             }
4200917           else if (format[f] == 'c')
4200918             {
4200919               // ---------------------- unsigned char.
4200920               value_ui = va_arg (ap, unsigned int);
4200921               string[s] = (char) value_ui;
4200922               s += 1;
4200923               f += 1;
4200924             }
4200925           else if (format[f] == 's')
4200926             {
4200927               // --------------------------- string.
4200928               value_cp = va_arg (ap, char *);
4200929               filler = ' ';
4200930
4200931               s +=
4200932                 strtostr_fill (value_cp, &string[s],
4200933                                alignment, filler, remain);
4200934               f += 1;
4200935             }
4200936           else
4200937             {
4200938               // ------------------- unsupported or
4200939               // unknown specifier.
4200940               ;
4200941             }
4200942           // ----------------------------------------
4200943           // End of specifier.
4200944           // ----------------------------------------
4200945           width_string[0] = '\0';
4200946           precision_string[0] = '\0';
4200947
4200948           specifier = 0;
4200949           specifier_flags = 0;
4200950           specifier_width = 0;
4200951           specifier_precision = 0;
4200952           specifier_type = 0;
4200953
4200954           flag_plus = 0;
4200955           flag_minus = 0;
```



```
4200956           flag_space = 0;
4200957           flag_alternate = 0;
4200958           flag_zero = 0;
4200959         }
4200960     }
4200961   string[s] = '\0';
4200962   return s;
4200963 }
4200964
4200965 //---------------------------------------------------
4200966 // Static functions.
4200967 //---------------------------------------------------
4200968 static size_t
4200969 uimaxtoa (uintmax_t integer, char *buffer, int base,
4200970           int uppercase, size_t size)
4200971 {
4200972   // ------------------------------------------
4200973   // Convert a maximum rank integer into a string.
4200974   // ------------------------------------------
4200975
4200976   uintmax_t integer_copy = integer;
4200977   size_t digits;
4200978   int b;
4200979   unsigned char remainder;
4200980
4200981   for (digits = 0; integer_copy > 0; digits++)
4200982     {
4200983       integer_copy = integer_copy / base;
4200984     }
4200985
4200986   if (buffer == NULL && integer == 0)
4200987     return 1;
4200988   if (buffer == NULL && integer > 0)
4200989     return digits;
4200990
4200991   if (integer == 0)
4200992     {
4200993       buffer[0] = '0';
4200994       buffer[1] = '\0';
4200995       return 1;
4200996     }
4200997   //
4200998   // Fix the maximum number of digits.
4200999   //
4201000   if (size > 0 && digits > size)
4201001     digits = size;
4201002   //
4201003   *(buffer + digits) = '\0';    // End of string.
4201004
4201005   for (b = digits - 1; integer != 0 && b >= 0; b--)
4201006     {
4201007       remainder = integer % base;
4201008       integer = integer / base;
4201009
4201010       if (remainder <= 9)
4201011         {
4201012           *(buffer + b) = remainder + '0';
4201013         }
4201014       else
4201015         {
4201016           if (uppercase)
4201017             {
4201018               *(buffer + b) = remainder - 10 + 'A';
4201019             }
4201020           else
4201021             {
4201022               *(buffer + b) = remainder - 10 + 'a';
4201023             }
4201024         }
4201025     }
4201026   return digits;
4201027 }
4201028
4201029 //---------------------------------------------------
4201030 static size_t
4201031 imaxtoa (intmax_t integer, char *buffer, int base,
4201032          int uppercase, size_t size)
4201033 {
4201034   // ------------------------------------------
4201035   // Convert a maximum rank integer with sign into a
4201036   // string.
4201037   // ------------------------------------------
4201038
4201039   if (integer >= 0)
4201040     {
4201041       return uimaxtoa (integer, buffer, base,
4201042                        uppercase, size);
```

```
    }
  //
  // At this point, there is a negative number, less
  // than zero.
  //
  if (buffer == NULL)
    {
      return uimaxtoa (-integer, NULL, base, uppercase,
                       size) + 1;
    }

  *buffer = '-';        // The minus sign is needed at
  // the beginning.
  if (size == 1)
    {
      *(buffer + 1) = '\0';
      return 1;
    }
  else
    {
      return uimaxtoa (-integer, buffer + 1, base,
                       uppercase, size - 1) + 1;
    }
}

//-------------------------------------------------------
static size_t
simaxtoa (intmax_t integer, char *buffer, int base,
          int uppercase, size_t size)
{
  // -----------------------------------------------
  // Convert a maximum rank integer with sign into a
  // string, placing
  // the sign also if it is positive.
  // -----------------------------------------------

  if (buffer == NULL && integer >= 0)
    {
      return uimaxtoa (integer, NULL, base, uppercase,
                       size) + 1;
    }

  if (buffer == NULL && integer < 0)
    {
      return uimaxtoa (-integer, NULL, base, uppercase,
                       size) + 1;
    }
  //
  // At this point, 'buffer' is different from NULL.
  //
  if (integer >= 0)
    {
      *buffer = '+';
    }
  else
    {
      *buffer = '-';
    }

  if (size == 1)
    {
      *(buffer + 1) = '\0';
      return 1;
    }

  if (integer >= 0)
    {
      return uimaxtoa (integer, buffer + 1, base,
                       uppercase, size - 1) + 1;
    }
  else
    {
      return uimaxtoa (-integer, buffer + 1, base,
                       uppercase, size - 1) + 1;
    }
}

//-------------------------------------------------------
static size_t
uimaxtoa_fill (uintmax_t integer, char *buffer,
               int base, int uppercase, int width,
               int filler, int max)
{
  // -----------------------------------------------
  // Convert a maximum rank integer without sign into
  // a string,
  // takeing care of the alignment.
```

```
  // -----------------------------------------------

  size_t size_i;
  size_t size_f;

  if (max < 0)
    return 0;   // «max» deve essere un valore
  // positivo.

  size_i = uimaxtoa (integer, NULL, base, uppercase, 0);

  if (width > 0 && max > 0 && width > max)
    width = max;
  if (width < 0 && -max < 0 && width < -max)
    width = -max;

  if (size_i > abs (width))
    {
      return uimaxtoa (integer, buffer, base,
                       uppercase, abs (width));
    }

  if (width == 0 && max > 0)
    {
      return uimaxtoa (integer, buffer, base,
                       uppercase, max);
    }

  if (width == 0)
    {
      return uimaxtoa (integer, buffer, base,
                       uppercase, abs (width));
    }
  //
  // size_i <= abs (width).
  //
  size_f = abs (width) - size_i;

  if (width < 0)
    {
      // Left alignment.
      uimaxtoa (integer, buffer, base, uppercase, 0);
      memset (buffer + size_i, filler, size_f);
    }
  else
    {
      // Right alignment.
      memset (buffer, filler, size_f);
      uimaxtoa (integer, buffer + size_f, base,
                uppercase, 0);
    }
  *(buffer + abs (width)) = '\0';

  return abs (width);
}

//-------------------------------------------------------
static size_t
imaxtoa_fill (intmax_t integer, char *buffer, int base,
              int uppercase, int width, int filler, int max)
{
  // -----------------------------------------------
  // Convert a maximum rank integer with sign into a
  // string,
  // takeing care of the alignment.
  // -----------------------------------------------

  size_t size_i;
  size_t size_f;

  if (max < 0)
    return 0;   // 'max' must be a positive value.

  size_i = imaxtoa (integer, NULL, base, uppercase, 0);

  if (width > 0 && max > 0 && width > max)
    width = max;
  if (width < 0 && -max < 0 && width < -max)
    width = -max;

  if (size_i > abs (width))
    {
      return imaxtoa (integer, buffer, base, uppercase,
                      abs (width));
    }

  if (width == 0 && max > 0)
```

```
4201217     {
4201218       return imaxtoa (integer, buffer, base, uppercase,
4201219                       max);
4201220     }
4201221
4201222   if (width == 0)
4201223     {
4201224       return imaxtoa (integer, buffer, base, uppercase,
4201225                       abs (width));
4201226     }
4201227
4201228   // size_i <= abs (width).
4201229
4201230   size_f = abs (width) - size_i;
4201231
4201232   if (width < 0)
4201233     {
4201234       // Left alignment.
4201235       imaxtoa (integer, buffer, base, uppercase, 0);
4201236       memset (buffer + size_i, filler, size_f);
4201237     }
4201238   else
4201239     {
4201240       // Right alignment.
4201241       memset (buffer, filler, size_f);
4201242       imaxtoa (integer, buffer + size_f, base,
4201243                uppercase, 0);
4201244     }
4201245   *(buffer + abs (width)) = '\0';
4201246
4201247   return abs (width);
4201248 }
4201249
4201250 //-------------------------------------------------------
4201251 static size_t
4201252 simaxtoa_fill (intmax_t integer, char *buffer,
4201253                int base, int uppercase, int width,
4201254                int filler, int max)
4201255 {
4201256   // -----------------------------------------------
4201257   // Convert a maximum rank integer with sign into a
4201258   // string,
4201259   // placing the sign also if it is positive and
4201260   // takeing care of the
4201261   // alignment.
4201262   // -----------------------------------------------
4201263
4201264   size_t size_i;
4201265   size_t size_f;
4201266
4201267   if (max < 0)
4201268     return 0;   // 'max' must be a positive value.
4201269
4201270   size_i = simaxtoa (integer, NULL, base, uppercase, 0);
4201271
4201272   if (width > 0 && max > 0 && width > max)
4201273     width = max;
4201274   if (width < 0 && -max < 0 && width < -max)
4201275     width = -max;
4201276
4201277   if (size_i > abs (width))
4201278     {
4201279       return simaxtoa (integer, buffer, base,
4201280                        uppercase, abs (width));
4201281     }
4201282
4201283   if (width == 0 && max > 0)
4201284     {
4201285       return simaxtoa (integer, buffer, base,
4201286                        uppercase, max);
4201287     }
4201288
4201289   if (width == 0)
4201290     {
4201291       return simaxtoa (integer, buffer, base,
4201292                        uppercase, abs (width));
4201293     }
4201294   //
4201295   // size_i <= abs (width).
4201296   //
4201297   size_f = abs (width) - size_i;
4201298
4201299   if (width < 0)
4201300     {
4201301       // Left alignment.
4201302       simaxtoa (integer, buffer, base, uppercase, 0);
4201303       memset (buffer + size_i, filler, size_f);
```



```
4201304     }
4201305   else
4201306     {
4201307       // Right alignment.
4201308       memset (buffer, filler, size_f);
4201309       simaxtoa (integer, buffer + size_f, base,
4201310                 uppercase, 0);
4201311     }
4201312   *(buffer + abs (width)) = '\0';
4201313
4201314   return abs (width);
4201315 }
4201316
4201317 //-------------------------------------------------------
4201318 static size_t
4201319 strtostr_fill (char *string, char *buffer, int width,
4201320                int filler, int max)
4201321 {
4201322   // -----------------------------------------------
4201323   // Transfer a string with care for the alignment.
4201324   // -----------------------------------------------
4201325
4201326   size_t size_s;
4201327   size_t size_f;
4201328
4201329   if (max < 0)
4201330     return 0;   // 'max' must be a positive value.
4201331
4201332   size_s = strlen (string);
4201333
4201334   if (width > 0 && max > 0 && width > max)
4201335     width = max;
4201336   if (width < 0 && -max < 0 && width < -max)
4201337     width = -max;
4201338
4201339   if (width != 0 && size_s > abs (width))
4201340     {
4201341       memcpy (buffer, string, abs (width));
4201342       buffer[width] = '\0';
4201343       return width;
4201344     }
4201345
4201346   if (width == 0 && max > 0 && size_s > max)
4201347     {
4201348       memcpy (buffer, string, max);
4201349       buffer[max] = '\0';
4201350       return max;
4201351     }
4201352
4201353   if (width == 0 && max > 0 && size_s < max)
4201354     {
4201355       memcpy (buffer, string, size_s);
4201356       buffer[size_s] = '\0';
4201357       return size_s;
4201358     }
4201359   //
4201360   // width =! 0
4201361   // size_s <= abs (width)
4201362   //
4201363   size_f = abs (width) - size_s;
4201364
4201365   if (width < 0)
4201366     {
4201367       // Right alignment.
4201368       memset (buffer, filler, size_f);
4201369       strncpy (buffer + size_f, string, size_s);
4201370     }
4201371   else
4201372     {
4201373       // Left alignment.
4201374       strncpy (buffer, string, size_s);
4201375       memset (buffer + size_s, filler, size_f);
4201376     }
4201377   *(buffer + abs (width)) = '\0';
4201378
4201379   return abs (width);
4201380 }
```

95.18.43 lib/stdio/vsprintf.c

Si veda la sezione 88.137.

```
4210001 #include <stdio.h>
4210002 //-------------------------------------------------------
4210003 int
4210004 vsprintf (char *restrict string,
4210005           const char *restrict format, va_list arg)
```

```
4210006 {
4210007   return (vsnprintf (string, BUFSIZ, format, arg));
4210008 }
```

95.18.44 lib/stdio/vsscanf.c

«

Si veda la sezione 88.138.

```
4220001 #include <stdio.h>
4220002
4220003 //-----------------------------------------------------------
4220004 int vfsscanf (FILE * restrict fp, const char *string,
4220005               const char *restrict format, va_list ap);
4220006 //-----------------------------------------------------------
4220007 int
4220008 vsscanf (const char *string,
4220009         const char *restrict format, va_list ap)
4220010 {
4220011   return (vfsscanf (NULL, string, format, ap));
4220012 }
4220013
4220014 //-----------------------------------------------------------
```

95.19 os32: «lib/stdlib.h»

«

Si veda la sezione 91.3.

```
4230001 #ifndef _STDLIB_H
4230002 #define _STDLIB_H       1
4230003 //-----------------------------------------------------------
4230004 #include <size_t.h>
4230005 #include <wchar_t.h>
4230006 #include <NULL.h>
4230007 #include <limits.h>
4230008 #include <restrict.h>
4230009 #include <stdint.h>
4230010 //-----------------------------------------------------------
4230011 typedef struct
4230012 {
4230013   int quot;
4230014   int rem;
4230015 } div_t;
4230016 //-----------------------------------------------------------
4230017 typedef struct
4230018 {
4230019   long int quot;
4230020   long int rem;
4230021 } ldiv_t;
4230022 //-----------------------------------------------------------
4230023 typedef struct
4230024 {
4230025   long long int quot;
4230026   long long int rem;
4230027 } lldiv_t;
4230028 //-----------------------------------------------------------
4230029 typedef void (*atexit_t) (void);        // Non standard.
4230030                                         // [1]
4230031 //
4230032 // [1] The type 'atexit_t' is a pointer to a function
4230033 //     for the "at exit" procedure, with no parameters
4230034 //     and returning void. With the declaration of type
4230035 //     'atexit_t', the function prototype of 'atexit()'
4230036 //     is easier to declare and to understand. Original
4230037 //     declaration is:
4230038 //
4230039 //     int atexit (void (*function) (void));
4230040 //
4230041 //-----------------------------------------------------------
4230042 typedef struct
4230043 {
4230044   uintptr_t allocated:1, filler:1, next:30;
4230045 } _alloc_head_t;        // Non standard [2]
4230046 //
4230047 // [2] This is used for the 'malloc()' management, as
4230048 //     the pointer to the following element of memory,
4230049 //     that might be free or allocated.
4230050 //
4230051 // La dimensione di «uintptr_t» condiziona la struttura
4230052 // «mm_head_t» e la dimensione delle unità minime di
4230053 // memoria allocata. «uintptr_t» è da 32 bit, così
4230054 // l'immagine del kernel è allineata a blocchi da
4230055 // 32 bit e così deve essere anche per gli altri
4230056 // blocchi di memoria.
4230057 // Essendo i blocchi di memoria multipli di 32 bit, gli
4230058 // indirizzi sono sempre multipli di 4 (4 byte);
4230059 // pertanto, servono solo 30 bit per rappresentare
4230060 // l'indirizzo, che poi viene ottenuto moltiplicandolo
```



```
4230061 // per quattro. Di conseguenza, il bit meno
4230062 // significativo viene usato per annotare se il blocco
4230063 // di memoria è libero e il bit successivo non viene
4230064 // usato. Questo meccanismo potrebbe essere usato anche
4230065 // con un indirizzamento a 16 bit, dove servirebbero 15
4230066 // bit per indirizzi multipli di due byte.
4230067 //
4230068 //-----------------------------------------------------------
4230069 #define EXIT_FAILURE    1
4230070 #define EXIT_SUCCESS    0
4230071 #define RAND_MAX        INT_MAX
4230072 #define MB_CUR_MAX      ((size_t) MB_LEN_MAX)
4230073 //-----------------------------------------------------------
4230074 void _Exit (int status);
4230075 void abort (void);
4230076 int abs (int j);
4230077 int atexit (atexit_t function);
4230078 int atoi (const char *string);
4230079 long int atol (const char *string);
4230080 #define  calloc(b, s) (malloc ((b) * (s)))
4230081 div_t div (int numer, int denom);
4230082 void exit (int status);
4230083 void free (void *ptr);
4230084 char *getenv (const char *name);
4230085 long int labs (long int j);
4230086 long long int llabs (long long int j);
4230087 ldiv_t ldiv (long int numer, long int denom);
4230088 lldiv_t lldiv (long long int numer, long long int denom);
4230089 void *malloc (size_t size);
4230090 int putenv (const char *string);
4230091 void qsort (void *base, size_t nmemb, size_t size,
4230092             int (*compare) (const void *, const void *));
4230093 int rand (void);
4230094 void *realloc (void *ptr, size_t size);
4230095 int setenv (const char *name, const char *value,
4230096             int overwrite);
4230097 void srand (unsigned int seed);
4230098 long int strtol (const char *restrict string,
4230099                  char **restrict endptr, int base);
4230100 unsigned long int strtoul (const char *restrict string,
4230101                            char **restrict endptr,
4230102                            int base);
4230103 //int             system   (const char *string);
4230104 int unsetenv (const char *name);
4230105 //-----------------------------------------------------------
4230106 #endif
```


95.19.1 lib/stdlib/_Exit.c 846
95.19.2 lib/stdlib/abort.c 846
95.19.3 lib/stdlib/abs.c 846
95.19.4 lib/stdlib/atexit.c 847
95.19.5 lib/stdlib/atoi.c 847
95.19.6 lib/stdlib/atol.c 848
95.19.7 lib/stdlib/div.c 848
95.19.8 lib/stdlib/environment.c 848
95.19.9 lib/stdlib/exit.c 849
95.19.10 lib/stdlib/getenv.c 850
95.19.11 lib/stdlib/labs.c 851
95.19.12 lib/stdlib/ldiv.c 851
95.19.13 lib/stdlib/llabs.c 851
95.19.14 lib/stdlib/lldiv.c 851
95.19.15 lib/stdlib/putenv.c 851
95.19.16 lib/stdlib/qsort.c 853
95.19.17 lib/stdlib/rand.c 854
95.19.18 lib/stdlib/setenv.c 855
95.19.19 lib/stdlib/strtol.c 856
95.19.20 lib/stdlib/strtoul.c 859
95.19.21 lib/stdlib/unsetenv.c 859
95.19.22 lib/stdlib_alloc/_alloc_list.c 860
95.19.23 lib/stdlib_alloc/free.c 861

95.19.24 lib/stdlib_alloc/malloc.c 862
95.19.25 lib/stdlib_alloc/realloc.c 865

95.19.1 lib/stdlib/_Exit.c

Si veda la sezione 87.2.

```
4240001 #include <stdlib.h>
4240002 #include <sys/os32.h>
4240003 //-----------------------------------------------------------
4240004 void
4240005 _Exit (int status)
4240006 {
4240007   sysmsg_exit_t msg;
4240008   //
4240009   // Only the low eight bit are returned.
4240010   //
4240011   msg.status = (status & 0xFF);
4240012   //
4240013   //
4240014   //
4240015   sys (SYS_EXIT, &msg, (sizeof msg));
4240016   //
4240017   // Should not return from system call, but if it
4240018   // does, loop
4240019   // forever:
4240020   //
4240021   while (1);
4240022 }
```

95.19.2 lib/stdlib/abort.c

Si veda la sezione 88.2.

```
4250001 #include <stdlib.h>
4250002 #include <sys/types.h>
4250003 #include <signal.h>
4250004 #include <unistd.h>
4250005 //-----------------------------------------------------------
4250006 void
4250007 abort (void)
4250008 {
4250009   pid_t pid;
4250010   sighandler_t sig_previous;
4250011   //
4250012   // Set 'SIGABRT' to a default action.
4250013   //
4250014   sig_previous = signal (SIGABRT, SIG_DFL);
4250015   //
4250016   // If the previous action was something different
4250017   // than symbolic
4250018   // ones, configure again the previous action.
4250019   //
4250020   if (sig_previous != SIG_DFL &&
4250021       sig_previous != SIG_IGN && sig_previous != SIG_ERR)
4250022     {
4250023       signal (SIGABRT, sig_previous);
4250024     }
4250025   //
4250026   // Get current process ID and sent the signal.
4250027   //
4250028   pid = getpid ();
4250029   kill (pid, SIGABRT);
4250030   //
4250031   // Second chance
4250032   //
4250033   for (;;)
4250034     {
4250035       signal (SIGABRT, SIG_DFL);
4250036       pid = getpid ();
4250037       kill (pid, SIGABRT);
4250038     }
4250039 }
```

95.19.3 lib/stdlib/abs.c

Si veda la sezione 88.3.

```
4260001 #include <stdlib.h>
4260002 //-----------------------------------------------------------
4260003 int
4260004 abs (int j)
4260005 {
4260006   if (j < 0)
4260007     {
4260008       return -j;
4260009     }
4260010   else
4260011     {
4260012       return j;
4260013     }
4260014 }
```


95.19.4 lib/stdlib/atexit.c

Si veda la sezione 88.7.

```
4270001 #include <stdlib.h>
4270002 //-----------------------------------------------------------
4270003 atexit_t _atexit_table[ATEXIT_MAX];
4270004 //-----------------------------------------------------------
4270005 void
4270006 _atexit_setup (void)
4270007 {
4270008   int a;
4270009   //
4270010   for (a = 0; a < ATEXIT_MAX; a++)
4270011     {
4270012       _atexit_table[a] = NULL;
4270013     }
4270014 }
4270015
4270016 //-----------------------------------------------------------
4270017 int
4270018 atexit (atexit_t function)
4270019 {
4270020   int a;
4270021   //
4270022   if (function == NULL)
4270023     {
4270024       return (-1);
4270025     }
4270026   //
4270027   for (a = 0; a < ATEXIT_MAX; a++)
4270028     {
4270029       if (_atexit_table[a] == NULL)
4270030         {
4270031           _atexit_table[a] = function;
4270032           return (0);
4270033         }
4270034     }
4270035   //
4270036   return (-1);
4270037 }
```

95.19.5 lib/stdlib/atoi.c

Si veda la sezione 88.8.

```
4280001 #include <stdlib.h>
4280002 #include <ctype.h>
4280003 //-----------------------------------------------------------
4280004 int
4280005 atoi (const char *string)
4280006 {
4280007   int i;
4280008   int sign = +1;
4280009   int number;
4280010   //
4280011   for (i = 0; isspace (string[i]); i++)
4280012     {
4280013       ;
4280014     }
4280015   //
4280016   if (string[i] == '+')
4280017     {
4280018       sign = +1;
4280019       i++;
4280020     }
4280021   else if (string[i] == '-')
4280022     {
4280023       sign = -1;
4280024       i++;
4280025     }
4280026   //
4280027   for (number = 0; isdigit (string[i]); i++)
4280028     {
4280029       number *= 10;
4280030       number += (string[i] - '0');
4280031     }
4280032   //
4280033   number *= sign;
4280034   //
4280035   return number;
```

```
4280036 }
```

95.19.6 lib/stdlib/atol.c

«

Si veda la sezione 88.8.

```
4290001 #include <stdlib.h>
4290002 #include <ctype.h>
4290003 //-----------------------------------------------------------
4290004 long int
4290005 atol (const char *string)
4290006 {
4290007   int i;
4290008   int sign = +1;
4290009   long int number;
4290010   //
4290011   for (i = 0; isspace (string[i]); i++)
4290012     {
4290013       ;
4290014     }
4290015   //
4290016   if (string[i] == '+')
4290017     {
4290018       sign = +1;
4290019       i++;
4290020     }
4290021   else if (string[i] == '-')
4290022     {
4290023       sign = -1;
4290024       i++;
4290025     }
4290026   //
4290027   for (number = 0; isdigit (string[i]); i++)
4290028     {
4290029       number *= 10;
4290030       number += (string[i] - '0');
4290031     }
4290032   //
4290033   number *= sign;
4290034   //
4290035   return number;
4290036 }
```

95.19.7 lib/stdlib/div.c

«

Si veda la sezione 88.17.

```
4300001 #include <stdlib.h>
4300002 //-----------------------------------------------------------
4300003 div_t
4300004 div (int numer, int denom)
4300005 {
4300006   div_t d;
4300007   d.quot = numer / denom;
4300008   d.rem = numer % denom;
4300009   return d;
4300010 }
```

95.19.8 lib/stdlib/environment.c

«

Si veda la sezione 91.1.

```
4310001 #include <stdlib.h>
4310002 #include <string.h>
4310003 //-----------------------------------------------------------
4310004 // This file contains a non standard definition,
4310005 // related to the environment handling.
4310006 //
4310007 // The file 'crt0.s', before calling the main function,
4310008 // calls the function '_environment_setup(), that is
4310009 // responsible for initializing the array
4310010 // '_environment_table[][]' and for copying the content
4310011 // of the environment, as it comes from the 'exec()'
4310012 // system call.
4310013 //
4310014 // The pointers to the environment strings organised
4310015 // inside the array '_environment_table[][]', are also
4310016 // copied inside the array of pointers
4310017 // '_environment[]'.
4310018 //
4310019 // After all that is done, inside 'crt0.s', the pointer
4310020 // to '_environment[]' is copied to the traditional
4310021 // variable 'environ' and also to the previous value of
4310022 // the pointer variable 'envp'.
4310023 //
4310024 // This way, applications will get the environment, but
4310025 // organised inside the table '_environment_table[][]'.
```



```
4310026 // So, functions like 'getenv()' and 'setenv()' do know
4310027 // where to look for.
4310028 //
4310029 // It is useful to notice that there is no prototype
4310030 // and no extern declaration inside the file
4310031 // <stdlib.h>, about this function and these arrays,
4310032 // because applications do not have to know about it.
4310033 //
4310034 // Please notice that 'environ' could be just the same
4310035 // as '_environment' here, but the common use puts
4310036 // 'environ' inside <unistd.h>, although for this
4310037 // implementation it should be better placed inside
4310038 // <stdlib.h>.
4310039 //
4310040 //-----------------------------------------------------------
4310041 char _environment_table[ARG_MAX / 32][ARG_MAX / 16];
4310042 char *_environment[ARG_MAX / 32 + 1];
4310043 //-----------------------------------------------------------
4310044 void
4310045 _environment_setup (char *envp[])
4310046 {
4310047   int e;
4310048   int s;
4310049   //
4310050   // Reset the '_environment_table[][]' array.
4310051   //
4310052   for (e = 0; e < ARG_MAX / 32; e++)
4310053     {
4310054       for (s = 0; s < ARG_MAX / 16; s++)
4310055         {
4310056           _environment_table[e][s] = 0;
4310057         }
4310058     }
4310059   //
4310060   // Set the '_environment[]' pointers. The final
4310061   // extra element must
4310062   // be a NULL pointer.
4310063   //
4310064   for (e = 0; e < ARG_MAX / 32; e++)
4310065     {
4310066       _environment[e] = _environment_table[e];
4310067     }
4310068   _environment[ARG_MAX / 32] = NULL;
4310069   //
4310070   // Copy the environment inside the array, but only
4310071   // if 'envp' is
4310072   // not NULL.
4310073   //
4310074   if (envp != NULL)
4310075     {
4310076       for (e = 0; envp[e] != NULL && e < ARG_MAX / 32; e++)
4310077         {
4310078           strncpy (_environment_table[e], envp[e],
4310079                    (ARG_MAX / 16) - 1);
4310080         }
4310081     }
4310082 }
```

95.19.9 lib/stdlib/exit.c

«

Si veda la sezione 88.7.

```
4320001 #include <stdlib.h>
4320002 #include <stdio.h>
4320003 //-----------------------------------------------------------
4320004 extern atexit_t _atexit_table[];
4320005 //-----------------------------------------------------------
4320006 void
4320007 exit (int status)
4320008 {
4320009   int a;
4320010   //
4320011   // The "at exit" functions must be called in reverse
4320012   // order.
4320013   //
4320014   for (a = (ATEXIT_MAX - 1); a >= 0; a--)
4320015     {
4320016       if (_atexit_table[a] != NULL)
4320017         {
4320018           (*_atexit_table[a]) ();
4320019         }
4320020     }
4320021   //
4320022   // Now: really exit.
4320023   //
4320024   _Exit (status);
4320025   //
```

```
4320026     // Should not return from system call, but if it
4320027     // does, loop
4320028     // forever:
4320029     //
4320030     while (1);
4320031   }
```

95.19.10 lib/stdlib/getenv.c

Si veda la sezione 88.52.

```
4330001   #include <stdlib.h>
4330002   #include <string.h>
4330003   //------------------------------------------------------------
4330004   extern char *_environment[];
4330005   //------------------------------------------------------------
4330006   char *
4330007   getenv (const char *name)
4330008   {
4330009     int e;        // First index: environment table
4330010     // items.
4330011     int f;        // Second index: environment string
4330012     // scan.
4330013     char *value;  // Pointer to the environment value
4330014     // found.
4330015     //
4330016     // Check if the input is valid. No error is
4330017     // reported.
4330018     //
4330019     if (name == NULL || strlen (name) == 0)
4330020       {
4330021         return (NULL);
4330022       }
4330023     //
4330024     // Scan the environment table items, with index 'e'.
4330025     // The pointer
4330026     // 'value' is initialized to NULL. If the pointer
4330027     // 'value' gets a
4330028     // valid pointer, the environment variable was found
4330029     // and a
4330030     // pointer to the beginning of its value is
4330031     // available.
4330032     //
4330033     for (value = NULL, e = 0; e < ARG_MAX / 32; e++)
4330034       {
4330035         //
4330036         // Scan the string of the environment item, with
4330037         // index 'f'.
4330038         // The scan continue until 'name[f]' and
4330039         // '_environment[e][f]'
4330040         // are equal.
4330041         //
4330042         for (f = 0;
4330043              f < ARG_MAX / 16 - 1
4330044              && name[f] == _environment[e][f]; f++)
4330045           {
4330046             ;     // Just scan.
4330047           }
4330048         //
4330049         // At this point, 'name[f]' and
4330050         // '_environment[e][f]' are
4330051         // different: if 'name[f]' is zero the name
4330052         // string is
4330053         // terminated; if '_environment[e][f]' is also
4330054         // equal to '=',
4330055         // the environment item is corresponding to the
4330056         // requested name.
4330057         //
4330058         if (name[f] == 0 && _environment[e][f] == '=')
4330059           {
4330060             //
4330061             // The pointer to the beginning of the
4330062             // environment value is
4330063             // calculated, and the external loop exit.
4330064             //
4330065             value = &_environment[e][f + 1];
4330066             break;
4330067           }
4330068       }
4330069     //
4330070     // The 'value' is returned: if it is still NULL,
4330071     // then, no
4330072     // environment variable with the requested name was
4330073     // found.
4330074     //
4330075     return (value);
4330076   }
```


95.19.11 lib/stdlib/labs.c

Si veda la sezione 88.3.

```
4340001   #include <stdlib.h>
4340002   //------------------------------------------------------------
4340003   long int
4340004   labs (long int j)
4340005   {
4340006     if (j < 0)
4340007       {
4340008         return -j;
4340009       }
4340010     else
4340011       {
4340012         return j;
4340013       }
4340014   }
```

95.19.12 lib/stdlib/ldiv.c

Si veda la sezione 88.17.

```
4350001   #include <stdlib.h>
4350002   //------------------------------------------------------------
4350003   ldiv_t
4350004   ldiv (long int numer, long int denom)
4350005   {
4350006     ldiv_t d;
4350007     d.quot = numer / denom;
4350008     d.rem = numer % denom;
4350009     return d;
4350010   }
```

95.19.13 lib/stdlib/llabs.c

Si veda la sezione 88.3.

```
4360001   #include <stdlib.h>
4360002   //------------------------------------------------------------
4360003   long long int
4360004   llabs (long long int j)
4360005   {
4360006     if (j < 0)
4360007       {
4360008         return -j;
4360009       }
4360010     else
4360011       {
4360012         return j;
4360013       }
4360014   }
```

95.19.14 lib/stdlib/lldiv.c

Si veda la sezione 88.17.

```
4370001   #include <stdlib.h>
4370002   //------------------------------------------------------------
4370003   lldiv_t
4370004   lldiv (long long int numer, long long int denom)
4370005   {
4370006     lldiv_t d;
4370007     d.quot = numer / denom;
4370008     d.rem = numer % denom;
4370009     return d;
4370010   }
```

95.19.15 lib/stdlib/putenv.c

Si veda la sezione 88.94.

```
4380001   #include <stdlib.h>
4380002   #include <string.h>
4380003   #include <errno.h>
4380004   //------------------------------------------------------------
4380005   extern char *_environment[];
4380006   //------------------------------------------------------------
4380007   int
4380008   putenv (const char *string)
4380009   {
4380010     int e;        // First index: environment table
4380011     // items.
4380012     int f;        // Second index: environment string
4380013     // scan.
4380014     //
4380015     // Check if the input is empty. No error is
4380016     // reported.
```

```
4380017     //
4380018     if (string == NULL || strlen (string) == 0)
4380019       {
4380020         return (0);
4380021       }
4380022     //
4380023     // Check if the input is valid: there must be a '='
4380024     // sign.
4380025     // Error here is reported.
4380026     //
4380027     if (strchr (string, '=') == NULL)
4380028       {
4380029         errset (EINVAL);  // Invalid argument.
4380030         return (-1);
4380031       }
4380032     //
4380033     // Scan the environment table items, with index 'e'.
4380034     // The intent is
4380035     // to find a previous environment variable with the
4380036     // same name.
4380037     //
4380038     for (e = 0; e < ARG_MAX / 32; e++)
4380039       {
4380040         //
4380041         // Scan the string of the environment item, with
4380042         // index 'f'.
4380043         // The scan continue until 'string[f]' and
4380044         // '_environment[e][f]'
4380045         // are equal.
4380046         //
4380047         for (f = 0;
4380048              f < ARG_MAX / 16 - 1
4380049              && string[f] == _environment[e][f]; f++)
4380050           {
4380051             ;     // Just scan.
4380052           }
4380053         //
4380054         // At this point, 'string[f-1]' and
4380055         // '_environment[e][f-1]'
4380056         // should contain '='. If it is so, the
4380057         // environment is replaced.
4380058         //
4380059         if (string[f - 1] == '='
4380060             && _environment[e][f - 1] == '=')
4380061           {
4380062             //
4380063             // The environment item was found: now
4380064             // replace the pointer.
4380065             //
4380066             _environment[e] = (char *) string;
4380067             //
4380068             // Return.
4380069             //
4380070             return (0);
4380071           }
4380072       }
4380073     //
4380074     // The item was not found. Scan again for a free
4380075     // slot.
4380076     //
4380077     for (e = 0; e < ARG_MAX / 32; e++)
4380078       {
4380079         if (_environment[e] == NULL
4380080             || _environment[e][0] == 0)
4380081           {
4380082             //
4380083             // An empty item was found and the pointer
4380084             // will be
4380085             // replaced.
4380086             //
4380087             _environment[e] = (char *) string;
4380088             //
4380089             // Return.
4380090             //
4380091             return (0);
4380092           }
4380093       }
4380094     //
4380095     // Sorry: the empty slot was not found!
4380096     //
4380097     errset (ENOMEM);      // Not enough space.
4380098     return (-1);
4380099 }
```


95.19.16 lib/stdlib/qsort.c

«

Si veda la sezione 88.96.

```
4390001 #include <stdlib.h>
4390002 #include <string.h>
4390003 #include <errno.h>
4390004 //-------------------------------------------------------
4390005 static int part (char *array, size_t size, int a,
4390006                  int z, int (*compare) (const void *,
4390007                                         const void *));
4390008 static void sort (char *array, size_t size, int a,
4390009                   int z, int (*compare) (const void *,
4390010                                          const void *));
4390011 //-------------------------------------------------------
4390012 void
4390013 qsort (void *base, size_t nmemb, size_t size,
4390014        int (*compare) (const void *, const void *))
4390015 {
4390016   if (size <= 1)
4390017     {
4390018       //
4390019       // There is nothing to sort!
4390020       //
4390021       return;
4390022     }
4390023   else
4390024     {
4390025       sort ((char *) base, size, 0, (int) (nmemb - 1),
4390026             compare);
4390027     }
4390028 }
4390029
4390030 //-------------------------------------------------------
4390031 static void
4390032 sort (char *array, size_t size, int a, int z,
4390033       int (*compare) (const void *, const void *))
4390034 {
4390035   int loc;
4390036   //
4390037   if (z > a)
4390038     {
4390039       loc = part (array, size, a, z, compare);
4390040       if (loc >= 0)
4390041         {
4390042           sort (array, size, a, loc - 1, compare);
4390043           sort (array, size, loc + 1, z, compare);
4390044         }
4390045     }
4390046 }
4390047
4390048 //-------------------------------------------------------
4390049 static int
4390050 part (char *array, size_t size, int a, int z,
4390051       int (*compare) (const void *, const void *))
4390052 {
4390053   int i;
4390054   int loc;
4390055   char *swap;
4390056   //
4390057   if (z <= a)
4390058     {
4390059       errset (EUNKNOWN);        // Should never
4390060                                 // happen.
4390061       return (-1);
4390062     }
4390063   //
4390064   // Index 'i' after the first element; index 'loc' at
4390065   // the last
4390066   // position.
4390067   //
4390068   i = a + 1;
4390069   loc = z;
4390070   //
4390071   // Prepare space in memory for element swap.
4390072   //
4390073   swap = malloc (size);
4390074   if (swap == NULL)
4390075     {
4390076       errset (ENOMEM);
4390077       return (-1);
4390078     }
4390079   //
4390080   // Loop as long as index 'loc' is higher than index
4390081   // 'i'.
4390082   // When index 'loc' is less or equal to index 'i',
4390083   // then, index 'loc' is the right position for the
4390084   // first element of the current piece of array.
4390085   //
```

```
4390086   for (;;)
4390087     {
4390088       //
4390089       // Index 'i' goes up...
4390090       //
4390091       for (; i < loc; i++)
4390092         {
4390093           if (compare
4390094               (&array[i * size], &array[a * size]) > 0)
4390095             {
4390096               break;
4390097             }
4390098         }
4390099       //
4390100       // Index 'loc' gose down...
4390101       //
4390102       for (;; loc--)
4390103         {
4390104           if (compare
4390105               (&array[loc * size], &array[a * size]) <= 0)
4390106             {
4390107               break;
4390108             }
4390109         }
4390110       //
4390111       // Swap elements related to index 'i' and 'loc'.
4390112       //
4390113       if (loc <= i)
4390114         {
4390115           //
4390116           // The array is completely scanned.
4390117           //
4390118           break;
4390119         }
4390120       else
4390121         {
4390122           memcpy (swap, &array[loc * size], size);
4390123           memcpy (&array[loc * size], &array[i * size],
4390124                   size);
4390125           memcpy (&array[i * size], swap, size);
4390126         }
4390127     }
4390128   //
4390129   // Swap the first element with the one related to
4390130   // the
4390131   // index 'loc'.
4390132   //
4390133   memcpy (swap, &array[loc * size], size);
4390134   memcpy (&array[loc * size], &array[a * size], size);
4390135   memcpy (&array[a * size], swap, size);
4390136   //
4390137   // Free the swap memory.
4390138   //
4390139   free (swap);
4390140   //
4390141   // Return the index 'loc'.
4390142   //
4390143   return (loc);
4390144 }
```

95.19.17 lib/stdlib/rand.c

«

Si veda la sezione 88.97.

```
4400001 #include <stdlib.h>
4400002 //-----------------------------------------------------------
4400003 static unsigned int _srand = 1; // The '_srand' rank
4400004                                 // must be at least
4400005                                 // 'unsigned int' and
4400006                                 // must be able to
4400007                                 // represent the value
4400008                                 // 'RAND_MAX'.
4400009 //-----------------------------------------------------------
4400010 int
4400011 rand (void)
4400012 {
4400013   _srand = _srand * 12345 + 123;
4400014   return _srand % ((unsigned int) RAND_MAX + 1);
4400015 }
4400016
4400017 //-----------------------------------------------------------
4400018 void
4400019 srand (unsigned int seed)
4400020 {
4400021   _srand = seed;
4400022 }
```


95.19.18 lib/stdlib/setenv.c

«

Si veda la sezione 88.104.

```
4410001 #include <stdlib.h>
4410002 #include <string.h>
4410003 #include <errno.h>
4410004 //-----------------------------------------------------------
4410005 extern char *_environment[];
4410006 extern char *_environment_table[];
4410007 //-----------------------------------------------------------
4410008 int
4410009 setenv (const char *name, const char *value, int overwrite)
4410010 {
4410011   int e;        // First index: environment table
4410012                 // items.
4410013   int f;        // Second index: environment string
4410014                 // scan.
4410015   //
4410016   // Check if the input is empty. No error is
4410017   // reported.
4410018   //
4410019   if (name == NULL || strlen (name) == 0)
4410020     {
4410021       return (0);
4410022     }
4410023   //
4410024   // Check if the input is valid: error here is
4410025   // reported.
4410026   //
4410027   if (strchr (name, '=') != NULL)
4410028     {
4410029       errset (EINVAL);  // Invalid argument.
4410030       return (-1);
4410031     }
4410032   //
4410033   // Check if the input is too big.
4410034   //
4410035   if ((strlen (name) + strlen (value) + 2) > ARG_MAX / 16)
4410036     {
4410037       //
4410038       // The environment to be saved is bigger than
4410039       // the
4410040       // available string size, inside
4410041       // '_environment_table[]'.
4410042       //
4410043       errset (ENOMEM);  // Not enough space.
4410044       return (-1);
4410045     }
4410046   //
4410047   // Scan the environment table items, with index 'e'.
4410048   // The intent is
4410049   // to find a previous environment variable with the
4410050   // same name.
4410051   //
4410052   for (e = 0; e < ARG_MAX / 32; e++)
4410053     {
4410054       //
4410055       // Scan the string of the environment item, with
4410056       // index 'f'.
4410057       // The scan continue until 'name[f]' and
4410058       // '_environment[e][f]'
4410059       // are equal.
4410060       //
4410061       for (f = 0;
4410062            f < ARG_MAX / 16 - 1
4410063            && name[f] == _environment[e][f]; f++)
4410064         {
4410065           ;     // Just scan.
4410066         }
4410067       //
4410068       // At this point, 'name[f]' and
4410069       // '_environment[e][f]' are
4410070       // different: if 'name[f]' is zero the name
4410071       // string is
4410072       // terminated; if '_environment[e][f]' is also
4410073       // equal to '=',
4410074       // the environment item is corresponding to the
4410075       // requested name.
4410076       //
4410077       if (name[f] == 0 && _environment[e][f] == '=')
4410078         {
4410079           //
4410080           // The environment item was found; if it can
4410081           // be overwritten,
4410082           // the write is done.
4410083           //
4410084           if (overwrite)
4410085             {
```

```
4410086                 //
4410087                 // To be able to handle both 'setenv()'
4410088                 // and 'putenv()',
4410089                 // before removing the item, it is fixed
4410090                 // the pointer to
4410091                 // the global environment table.
4410092                 //
4410093                 _environment[e] = _environment_table[e];
4410094                 //
4410095                 // Now copy the new environment. The
4410096                 // string size was
4410097                 // already checked.
4410098                 //
4410099                 strcpy (_environment[e], name);
4410100                 strcat (_environment[e], "=");
4410101                 strcat (_environment[e], value);
4410102                 //
4410103                 // Return.
4410104                 //
4410105                 return (0);
4410106               }
4410107             //
4410108             // Cannot overwrite!
4410109             //
4410110             errset (EUNKNOWN);
4410111             return (-1);
4410112           }
4410113       }
4410114     //
4410115     // The item was not found. Scan again for a free
4410116     // slot.
4410117     //
4410118     for (e = 0; e < ARG_MAX / 32; e++)
4410119       {
4410120         if (_environment[e] == NULL
4410121             || _environment[e][0] == 0)
4410122           {
4410123             //
4410124             // An empty item was found. To be able to
4410125             // handle both
4410126             // 'setenv()' and 'putenv()', it is fixed
4410127             // the pointer to
4410128             // the global environment table.
4410129             //
4410130             _environment[e] = _environment_table[e];
4410131             //
4410132             // Now copy the new environment. The string
4410133             // size was
4410134             // already checked.
4410135             //
4410136             strcpy (_environment[e], name);
4410137             strcat (_environment[e], "=");
4410138             strcat (_environment[e], value);
4410139             //
4410140             // Return.
4410141             //
4410142             return (0);
4410143           }
4410144       }
4410145     //
4410146     // Sorry: the empty slot was not found!
4410147     //
4410148     errset (ENOMEM);      // Not enough space.
4410149     return (-1);
4410150 }
```

95.19.19 lib/stdlib/strtol.c

«

Si veda la sezione 88.130.

```
4420001 #include <stdlib.h>
4420002 #include <ctype.h>
4420003 #include <errno.h>
4420004 #include <limits.h>
4420005 #include <stdbool.h>
4420006 //-------------------------------------------------------
4420007 #define isoctal(C)  ((int) (C >= '0' && C <= '7'))
4420008 //-------------------------------------------------------
4420009 long int
4420010 strtol (const char *restrict string,
4420011         char **restrict endptr, int base)
4420012 {
4420013   int i;
4420014   int sign = +1;
4420015   long int number;
4420016   long int previous;
4420017   int digit;
4420018   //
4420019   bool flag_prefix_oct = 0;
4420020   bool flag_prefix_exa = 0;
4420021   bool flag_prefix_dec = 0;
4420022   //
4420023   // Check base and string.
4420024   //
4420025   // With base 1 cannot do anything.
4420026   //
4420027   if (base < 0 || base > 36 || base == 1
4420028       || string == NULL || string[0] == 0)
4420029     {
4420030       if (endptr != NULL)
4420031         *endptr = (char *) string;
4420032       errset (EINVAL);  // Invalid argument.
4420033       return ((long int) 0);
4420034     }
4420035   //
4420036   // Eat initial spaces.
4420037   //
4420038   for (i = 0; isspace (string[i]); i++)
4420039     {
4420040       ;
4420041     }
4420042   //
4420043   // Check sign.
4420044   //
4420045   if (string[i] == '+')
4420046     {
4420047       sign = +1;
4420048       i++;
4420049     }
4420050   else if (string[i] == '-')
4420051     {
4420052       sign = -1;
4420053       i++;
4420054     }
4420055   //
4420056   // Check for prefix.
4420057   //
4420058   if (string[i] == '0')
4420059     {
4420060       if (string[i + 1] == 'x' || string[i + 1] == 'X')
4420061         {
4420062           flag_prefix_exa = 1;
4420063         }
4420064       else if (isoctal (string[i + 1]))
4420065         {
4420066           flag_prefix_oct = 1;
4420067         }
4420068       else
4420069         {
4420070           flag_prefix_dec = 1;
4420071         }
4420072     }
4420073   else if (isdigit (string[i]))
4420074     {
4420075       flag_prefix_dec = 1;
4420076     }
4420077   //
4420078   // Check compatibility with requested base.
4420079   //
4420080   if (flag_prefix_exa)
4420081     {
4420082       //
4420083       // At the moment, there is a zero and a 'x'.
4420084       // Might be
4420085       // exadecimal, or might be a number base 33 or
4420086       // more.
4420087       //
4420088       if (base == 0)
4420089         {
4420090           base = 16;
4420091         }
4420092       else if (base == 16)
4420093         {
4420094           ;     // Ok.
4420095         }
4420096       else if (base >= 33)
4420097         {
4420098           ;     // Ok.
4420099         }
4420100       else
4420101         {
4420102           //
4420103           // Incompatible sequence: only the initial
4420104           // zero is reported.
```

```
4420105           //
4420106           if (endptr != NULL)
4420107             *endptr = (char *) &string[i + 1];
4420108           return ((long int) 0);
4420109         }
4420110       //
4420111       // Move on, after the '0x' prefix.
4420112       //
4420113       i += 2;
4420114     }
4420115   //
4420116   if (flag_prefix_oct)
4420117     {
4420118       //
4420119       // There is a zero and a digit.
4420120       //
4420121       if (base == 0)
4420122         {
4420123           base = 8;
4420124         }
4420125       //
4420126       // Move on, after the '0' prefix.
4420127       //
4420128       i += 1;
4420129     }
4420130   //
4420131   if (flag_prefix_dec)
4420132     {
4420133       if (base == 0)
4420134         {
4420135           base = 10;
4420136         }
4420137     }
4420138   //
4420139   // Scan the string.
4420140   //
4420141   for (number = 0; string[i] != 0; i++)
4420142     {
4420143       if (string[i] >= '0' && string[i] <= '9')
4420144         {
4420145           digit = string[i] - '0';
4420146         }
4420147       else if (string[i] >= 'A' && string[i] <= 'Z')
4420148         {
4420149           digit = string[i] - 'A' + 10;
4420150         }
4420151       else if (string[i] >= 'a' && string[i] <= 'z')
4420152         {
4420153           digit = string[i] - 'a' + 10;
4420154         }
4420155       else
4420156         {
4420157           //
4420158           // This is an out of range digit.
4420159           //
4420160           digit = 999;
4420161         }
4420162       //
4420163       // Give a sign to the digit.
4420164       //
4420165       digit *= sign;
4420166       //
4420167       // Compare with the base.
4420168       //
4420169       if (base > (digit * sign))
4420170         {
4420171           //
4420172           // Check if the current digit can be safely
4420173           // computed.
4420174           //
4420175           previous = number;
4420176           number *= base;
4420177           number += digit;
4420178           if (number / base != previous)
4420179             {
4420180               //
4420181               // Out of range.
4420182               //
4420183               if (endptr != NULL)
4420184                 *endptr = (char *) &string[i + 1];
4420185               errset (ERANGE);  // Result too large.
4420186               if (sign > 0)
4420187                 {
4420188                   return (LONG_MAX);
4420189                 }
4420190               else
4420191                 {
```



```
4420192                   return (LONG_MIN);
4420193                 }
4420194             }
4420195         }
4420196       else
4420197         {
4420198           if (endptr != NULL)
4420199             *endptr = (char *) &string[i];
4420200           return (number);
4420201         }
4420202     }
4420203   //
4420204   // The string is finished.
4420205   //
4420206   if (endptr != NULL)
4420207     *endptr = (char *) &string[i];
4420208   //
4420209   return (number);
4420210 }
```

95.19.20 lib/stdlib/strtoul.c

Si veda la sezione 88.130.

```
4430001 #include <stdlib.h>
4430002 #include <ctype.h>
4430003 #include <errno.h>
4430004 #include <limits.h>
4430005 //-------------------------------------------------------------
4430006 // A really poor implementation. ,-(
4430007 //
4430008 unsigned long int
4430009 strtoul (const char *restrict string,
4430010          char **restrict endptr, int base)
4430011 {
4430012   return ((unsigned long int)
4430013           strtol (string, endptr, base));
4430014 }
```

95.19.21 lib/stdlib/unsetenv.c

Si veda la sezione 88.104.

```
4440001 #include <stdlib.h>
4440002 #include <string.h>
4440003 #include <errno.h>
4440004 //-------------------------------------------------------------
4440005 extern char *_environment[];
4440006 extern char *_environment_table[];
4440007 //-------------------------------------------------------------
4440008 int
4440009 unsetenv (const char *name)
4440010 {
4440011   int e;        // First index: environment table
4440012   // items.
4440013   int f;        // Second index: environment string
4440014   // scan.
4440015   //
4440016   // Check if the input is empty. No error is
4440017   // reported.
4440018   //
4440019   if (name == NULL || strlen (name) == 0)
4440020     {
4440021       return (0);
4440022     }
4440023   //
4440024   // Check if the input is valid: error here is
4440025   // reported.
4440026   //
4440027   if (strchr (name, '=') != NULL)
4440028     {
4440029       errset (EINVAL);  // Invalid argument.
4440030       return (-1);
4440031     }
4440032   //
4440033   // Scan the environment table items, with index 'e'.
4440034   //
4440035   for (e = 0; e < ARG_MAX / 32; e++)
4440036     {
4440037       //
4440038       // Scan the string of the environment item, with
4440039       // index 'f'.
4440040       // The scan continue until 'name[f]' and
4440041       // '_environment[e][f]'
4440042       // are equal.
4440043       //
4440044       for (f = 0;
```

```
4440045             f < ARG_MAX / 16 - 1
4440046             && name[f] == _environment[e][f]; f++)
4440047           {
4440048             ;     // Just scan.
4440049           }
4440050         //
4440051         // At this point, 'name[f]' and
4440052         // '_environment[e][f]' are
4440053         // different: if 'name[f]' is zero the name
4440054         // string is
4440055         // terminated; if '_environment[e][f]' is also
4440056         // equal to '=',
4440057         // the environment item is corresponding to the
4440058         // requested name.
4440059         //
4440060         if (name[f] == 0 && _environment[e][f] == '=')
4440061           {
4440062             //
4440063             // The environment item was found and it
4440064             // have to be removed.
4440065             // To be able to handle both 'setenv()' and
4440066             // 'putenv()',
4440067             // before removing the item, it is fixed the
4440068             // pointer to
4440069             // the global environment table.
4440070             //
4440071             _environment[e] = _environment_table[e];
4440072             //
4440073             // Now remove the environment item.
4440074             //
4440075             _environment[e][0] = 0;
4440076             break;
4440077           }
4440078       }
4440079     //
4440080     // Work done fine.
4440081     //
4440082     return (0);
4440083 }
```

95.19.22 lib/stdlib_alloc/_alloc_list.c

Si veda la sezione 88.76.

```
4450001 #include <stdlib.h>
4450002 #include <stdio.h>
4450003 #include <unistd.h>
4450004 #include <stdint.h>
4450005 //-------------------------------------------------------
4450006 extern uintptr_t _alloc_start;
4450007 //-------------------------------------------------------
4450008 void
4450009 _alloc_list (void)
4450010 {
4450011   uintptr_t start = _alloc_start;
4450012   uintptr_t end = (uintptr_t) sbrk (0);
4450013   _alloc_head_t *head = (void *) start;
4450014   size_t actual_size;
4450015   uintptr_t current;
4450016   uintptr_t next;
4450017   uintptr_t up_to;
4450018   int counter;
4450019   //
4450020   // Scandisce la lista di blocchi di memoria.
4450021   //
4450022   counter = 2;
4450023   while (counter)
4450024     {
4450025       //
4450026       // Annota la posizione attuale e quella
4450027       // successiva.
4450028       //
4450029       current = (uintptr_t) head;
4450030       next = head->next * (sizeof (_alloc_head_t));
4450031       if (next == start)
4450032         {
4450033           up_to = end;
4450034         }
4450035       else
4450036         {
4450037           up_to = next;
4450038         }
4450039       //
4450040       // Se è stato raggiunto il primo elemento,
4450041       // decrementa il
4450042       // contatore di una unità. Se è già a zero,
4450043       // esce.
```



```
4450044       //
4450045       if (current == start)
4450046         {
4450047           counter--;
4450048           if (counter == 0)
4450049             break;
4450050         }
4450051       //
4450052       // Determina la dimensione del blocco attuale.
4450053       //
4450054       if (current == start && next == start)
4450055         {
4450056           //
4450057           // Si tratta del primo e unico elemento
4450058           // della lista.
4450059           //
4450060           actual_size =
4450061             end - start - (sizeof (_alloc_head_t));
4450062         }
4450063       else
4450064         {
4450065           actual_size =
4450066             up_to - current - (sizeof (_alloc_head_t));
4450067         }
4450068       //
4450069       // Si mostra lo stato del blocco di memoria.
4450070       //
4450071       if (head->allocated)
4450072         {
4450073           printf ("[%s] used %08X..%08X size %08zX\n",
4450074                   __func__,
4450075                   current + (sizeof (_alloc_head_t)),
4450076                   up_to, actual_size);
4450077         }
4450078       else
4450079         {
4450080           printf ("[%s] free %08X..%08X size %08zX\n",
4450081                   __func__,
4450082                   current + (sizeof (_alloc_head_t)),
4450083                   up_to, actual_size);
4450084         }
4450085       //
4450086       // Si passa alla posizione successiva.
4450087       //
4450088       head = (void *) next;
4450089     }
4450090 }
```

95.19.23 lib/stdlib_alloc/free.c

Si veda la sezione 88.76.

```
4460001 #include <stdlib.h>
4460002 #include <stdio.h>
4460003 #include <unistd.h>
4460004 //-------------------------------------------------------
4460005 extern uintptr_t _alloc_start;
4460006 //-------------------------------------------------------
4460007 void
4460008 free (void *ptr)
4460009 {
4460010   _alloc_head_t *start = (_alloc_head_t *) _alloc_start;
4460011   _alloc_head_t *head_current = ((_alloc_head_t *) ptr) - 1;
4460012   _alloc_head_t *head_next;
4460013   //
4460014   // Verifica il blocco attuale e, se è possibile, lo
4460015   // libera.
4460016   //
4460017   if (head_current->allocated == 1)
4460018     {
4460019       head_current->allocated = 0;
4460020     }
4460021   else
4460022     {
4460023       printf ("[%s] ERROR: cannot free %08X!\n",
4460024               __func__,
4460025               (uintptr_t) head_current +
4460026               (sizeof (_alloc_head_t)));
4460027     }
4460028   //
4460029   // Scandisce i blocchi liberi, cercando quelli
4460030   // adiacenti per
4460031   // allungarli. Se il blocco successivo è il primo,
4460032   // termina,
4460033   // perché non può avvenire alcuna fusione con
4460034   // quello precedente.
4460035   //
```

```
4460036   head_current = start;
4460037   while (1)
4460038     {
4460039       //
4460040       // Individua il blocco successivo.
4460041       //
4460042       head_next =
4460043         (_alloc_head_t *) (head_current->next
4460044                            * (sizeof (_alloc_head_t)));
4460045       //
4460046       // Controlla se è il primo.
4460047       //
4460048       if (head_next == start)
4460049         {
4460050           break;
4460051         }
4460052       //
4460053       //
4460054       //
4460055       if (head_current->allocated == 0)
4460056         {
4460057           //
4460058           // Controlla se si può espandere.
4460059           //
4460060           if (head_next->allocated == 0)
4460061             {
4460062               head_current->next = head_next->next;
4460063             }
4460064           else
4460065             {
4460066               head_current = head_next;
4460067             }
4460068         }
4460069       else
4460070         {
4460071           head_current = (_alloc_head_t *)
4460072             (head_current->next * (sizeof (_alloc_head_t)));
4460073         }
4460074     }
4460075 }
```

95.19.24 lib/stdlib_alloc/malloc.c

«

Si veda la sezione 88.76.

```
4470001 #include <stdlib.h>
4470002 #include <unistd.h>
4470003 #include <errno.h>
4470004 //-------------------------------------------------------
4470005 uintptr_t _alloc_start = 0;
4470006 //-------------------------------------------------------
4470007 static int _alloc_init (void);
4470008 static void *_malloc (size_t size);
4470009 //-------------------------------------------------------
4470010 void *
4470011 malloc (size_t size)
4470012 {
4470013   void *pstatus;
4470014   int status;
4470015   //
4470016   // Verify to have initialized the allocation memory.
4470017   //
4470018   if (_alloc_start == 0)
4470019     {
4470020       status = _alloc_init ();
4470021       if (status < 0)
4470022         {
4470023           errset (ENOMEM);
4470024           return (NULL);
4470025         }
4470026     }
4470027   //
4470028   // Try to allocate as usual.
4470029   //
4470030   pstatus = _malloc (size);
4470031   //
4470032   if (pstatus == NULL)
4470033     {
4470034       //
4470035       // Try to increase memory for the process.
4470036       //
4470037       pstatus = sbrk (size);
4470038       if (pstatus == NULL)
4470039         {
4470040           //
4470041           // Sorry: no way to get memory.
4470042           //
4470043           errset (ENOMEM);
4470044           return (NULL);
4470045         }
4470046       //
4470047       // Ok. Now try again to allocate memory.
4470048       //
4470049       return (_malloc (size));
4470050     }
4470051   else
4470052     {
4470053       //
4470054       // The first allocation was successful.
4470055       //
4470056       return (pstatus);
4470057     }
4470058 }
4470059
4470060 //-------------------------------------------------------
4470061 static int
4470062 _alloc_init (void)
4470063 {
4470064   uintptr_t start;
4470065   uintptr_t end;
4470066   _alloc_head_t *head;
4470067   size_t available;
4470068   //
4470069   // Get size.
4470070   //
4470071   if (_alloc_start == 0)
4470072     {
4470073       _alloc_start = (uintptr_t) sbrk (0);
4470074     }
4470075   //
4470076   start = _alloc_start;
4470077   end = (uintptr_t) sbrk (0);
4470078   available = end - start;
4470079   //
4470080   // Check available space.
4470081   //
4470082   if (available < ((sizeof (_alloc_head_t)) * 2))
4470083     {
4470084       //
4470085       // Try to get a little memory.
4470086       //
4470087       sbrk ((sizeof (_alloc_head_t)) * 2);
4470088       end = (uintptr_t) sbrk (0);
4470089       available = end - start;
4470090       if (available < ((sizeof (_alloc_head_t)) * 2))
4470091         {
4470092           //
4470093           // Sorry!
4470094           //
4470095           return (-1);
4470096         }
4470097     }
4470098   //
4470099   // Prepare the list main node.
4470100   //
4470101   head = (_alloc_head_t *) start;
4470102   //
4470103   // Init the first free block, that points to itself,
4470104   // as it is
4470105   // the only one.
4470106   //
4470107   head->allocated = 0;
4470108   head->next = (start / (sizeof (_alloc_head_t)));
4470109   //
4470110   // Ok.
4470111   //
4470112   return (0);
4470113 }
4470114
4470115 //-------------------------------------------------------
4470116 static void *
4470117 _malloc (size_t size)
4470118 {
4470119   uintptr_t start = _alloc_start;
4470120   uintptr_t end = (uintptr_t) sbrk (0);
4470121   _alloc_head_t *head = (void *) start;
4470122   size_t actual_size;
4470123   uintptr_t current;
4470124   uintptr_t next;
4470125   uintptr_t new;
4470126   uintptr_t up_to;
4470127   int counter;
4470128   //
4470129   // Arrotonda in eccesso il valore di «size», in
```

```
4470130   // modo che sia un
4470131   // multiplo della dimensione di «_alloc_head_t».
4470132   // Altrimenti, la
4470133   // collocazione dei blocchi successivi può avvenire
4470134   // in modo
4470135   // non allineato.
4470136   //
4470137   size = (size + (sizeof (_alloc_head_t)) - 1);
4470138   size = size / (sizeof (_alloc_head_t));
4470139   size = size * (sizeof (_alloc_head_t));
4470140   //
4470141   // Cerca un blocco libero di dimensione sufficiente.
4470142   //
4470143   counter = 2;
4470144   while (counter)
4470145     {
4470146       //
4470147       // Annota la posizione attuale e quella
4470148       // successiva.
4470149       //
4470150       current = (uintptr_t) head;
4470151       next = head->next * (sizeof (_alloc_head_t));
4470152       //
4470153       if (next == start)
4470154         {
4470155           up_to = end;
4470156         }
4470157       else
4470158         {
4470159           up_to = next;
4470160         }
4470161       //
4470162       // Se è stato raggiunto il primo elemento,
4470163       // decrementa il
4470164       // contatore di una unità. Se è già a zero,
4470165       // esce.
4470166       //
4470167       if (current == start)
4470168         {
4470169           counter--;
4470170           if (counter == 0)
4470171             break;
4470172         }
4470173       //
4470174       // Controlla se si tratta di un blocco libero.
4470175       //
4470176
4470177       if (!head->allocated)
4470178         {
4470179           //
4470180           // Il blocco è libero: si deve determinarne
4470181           // la dimensione.
4470182           //
4470183           if (current == start && next == start)
4470184             {
4470185               //
4470186               // Si tratta del primo e unico elemento
4470187               // della lista.
4470188               //
4470189               actual_size =
4470190                 end - start - (sizeof (_alloc_head_t));
4470191             }
4470192           else
4470193             {
4470194               actual_size =
4470195                 up_to - current - (sizeof (_alloc_head_t));
4470196             }
4470197           //
4470198           // Si verifica che sia capiente.
4470199           //
4470200           if (actual_size >=
4470201               size + ((sizeof (_alloc_head_t)) * 2))
4470202             {
4470203               //
4470204               // C'è spazio per dividere il blocco.
4470205               //
4470206               new =
4470207                 current + size + (sizeof (_alloc_head_t));
4470208               //
4470209               // Aggiorna l'intestazione attuale.
4470210               //
4470211               head->allocated = 1;
4470212               head->next = new / (sizeof (_alloc_head_t));
4470213               //
4470214               // Predispone l'intestazione successiva.
4470215               //
4470216               head = (void *) new;
```



```
4470217               head->allocated = 0;
4470218               head->next = next / (sizeof (_alloc_head_t));
4470219               //
4470220               // Restituisce l'indirizzo iniziale
4470221               // dello spazio libero,
4470222               // successivo all'intestazione.
4470223               //
4470224               return (void *) (current +
4470225                                (sizeof (_alloc_head_t)));
4470226             }
4470227           else if (actual_size >= size)
4470228             {
4470229               //
4470230               // Il blocco va usato per intero.
4470231               //
4470232               head->allocated = 1;
4470233               //
4470234               // Restituisce l'indirizzo iniziale
4470235               // dello spazio libero,
4470236               // successivo all'intestazione.
4470237               //
4470238               return (void *) (current +
4470239                                (sizeof (_alloc_head_t)));
4470240             }
4470241         }
4470242       //
4470243       // Il blocco è allocato, oppure è di
4470244       // dimensione insufficiente;
4470245       // pertanto occorre passare alla posizione
4470246       // successiva.
4470247       //
4470248       head = (void *) next;
4470249     }
4470250   //
4470251   // Essendo terminato il ciclo precedente, vuol dire
4470252   // che non ci sono spazi disponibili.
4470253   //
4470254   errset (ENOMEM);
4470255   return NULL;
4470256 }
```

95.19.25 lib/stdlib_alloc/realloc.c

«

Si veda la sezione 88.76.

```
4480001 #include <stdlib.h>
4480002 #include <stdio.h>
4480003 #include <unistd.h>
4480004 #include <string.h>
4480005 //-------------------------------------------------------
4480006 extern uintptr_t _alloc_start;
4480007 //-------------------------------------------------------
4480008 void *
4480009 realloc (void *ptr, size_t size)
4480010 {
4480011   uintptr_t start = _alloc_start;
4480012   uintptr_t end = (uintptr_t) sbrk (0);
4480013   size_t actual_size;
4480014   _alloc_head_t *head = ((_alloc_head_t *) ptr) - 1;
4480015   _alloc_head_t *head_new;
4480016   void *ptr_new;
4480017   //
4480018   // Verifica che il puntatore riguardi effettivamente
4480019   // un'area occupata.
4480020   //
4480021   if (!head->allocated)
4480022     {
4480023       printf
4480024         ("[%s] ERROR: cannot re-allocate %08X that is "
4480025          "not already allocated!", __func__,
4480026          (uintptr_t) ptr);
4480027     }
4480028   //
4480029   // Arrotonda in eccesso il valore di «size», in
4480030   // modo che sia un
4480031   // multiplo della dimensione di «_alloc_head_t».
4480032   // Altrimenti, la
4480033   // collocazione dei blocchi successivi può avvenire
4480034   // in modo
4480035   // non allineato.
4480036   //
4480037   size = (size + (sizeof (_alloc_head_t)) - 1);
4480038   size = size / (sizeof (_alloc_head_t));
4480039   size = size * (sizeof (_alloc_head_t));
4480040   //
4480041   // Determina la dimensione attuale.
4480042   //
```

```
4480043   if ((head->next * (sizeof (_alloc_head_t))) == start)
4480044     {
4480045       actual_size = end - ((uintptr_t) ptr);
4480046     }
4480047   else
4480048     {
4480049       actual_size =
4480050         (head->next * (sizeof (_alloc_head_t))) -
4480051         ((uintptr_t) ptr);
4480052     }
4480053   //
4480054   // Se la dimensione richiesta è inferiore, può
4480055   // ridurre
4480056   // l'estensione del blocco.
4480057   //
4480058   if (size == actual_size)
4480059     {
4480060       return ptr;
4480061     }
4480062   else if (size <=
4480063            (actual_size - (sizeof (_alloc_head_t)) * 2))
4480064     {
4480065       //
4480066       // Si può ricavare lo spazio libero rimanente.
4480067       //
4480068       head_new = (_alloc_head_t *) (((char *) ptr) + size);
4480069       //
4480070       head_new->next = head->next;
4480071       head_new->allocated = 0;
4480072       //
4480073       head->next =
4480074         ((uintptr_t) head_new) / (sizeof (_alloc_head_t));
4480075       //
4480076       return ptr;
4480077     }
4480078   else if (size < actual_size)
4480079     {
4480080       //
4480081       // Anche se è minore, non si può ridurre lo
4480082       // spazio usato
4480083       // effettivamente.
4480084       //
4480085       return ptr;
4480086     }
4480087   else
4480088     {
4480089       //
4480090       // La dimensione richiesta è maggiore.
4480091       //
4480092       ptr_new = malloc (size);
4480093       //
4480094       if (ptr_new)
4480095         {
4480096           //
4480097           // Ricopia i dati nella nuova collocazione.
4480098           //
4480099           memcpy (ptr_new, ptr, actual_size);
4480100           //
4480101           // Libera la collocazione vecchia.
4480102           //
4480103           free (ptr);
4480104           //
4480105           return ptr_new;
4480106         }
4480107       else
4480108         {
4480109           return NULL;
4480110         }
4480111     }
4480112 }
```

95.20 os32: «lib/string.h»

«

Si veda la sezione 91.3.

```
4490001 #ifndef _STRING_H
4490002 #define _STRING_H       1
4490003 //---------------------------------------------------------
4490004 #include <size_t.h>
4490005 #include <NULL.h>
4490006 #include <restrict.h>
4490007 //---------------------------------------------------------
4490008 void *memccpy (void *restrict dst,
4490009                const void *restrict org, int c, size_t n);
4490010 void *memchr (const void *memory, int c, size_t n);
4490011 int memcmp (const void *memory1, const void *memory2,
4490012             size_t n);
```



```
4490013 void *memcpy (void *restrict dst,
4490014              const void *restrict org, size_t n);
4490015 void *memmove (void *dst, const void *org, size_t n);
4490016 void *memset (void *memory, int c, size_t n);
4490017 char *strcat (char *restrict dst, const char *restrict org);
4490018 char *strchr (const char *string, int c);
4490019 int strcmp (const char *string1, const char *string2);
4490020 int strcoll (const char *string1, const char *string2);
4490021 char *strcpy (char *restrict dst, const char *restrict org);
4490022 size_t strcspn (const char *string, const char *reject);
4490023 char *strdup (const char *string);
4490024 char *strerror (int errnum);
4490025 size_t strlen (const char *string);
4490026 char *strncat (char *restrict dst,
4490027               const char *restrict org, size_t n);
4490028 int strncmp (const char *string1, const char *string2,
4490029             size_t n);
4490030 char *strncpy (char *restrict dst,
4490031               const char *restrict org, size_t n);
4490032 char *strpbrk (const char *string, const char *accept);
4490033 char *strrchr (const char *string, int c);
4490034 size_t strspn (const char *string, const char *accept);
4490035 char *strstr (const char *string, const char *substring);
4490036 char *strtok (char *restrict string,
4490037              const char *restrict delim);
4490038 size_t strxfrm (char *restrict dst,
4490039                 const char *restrict org, size_t n);
4490040 //---------------------------------------------------------
4490041
4490042 #endif
```


95.20.1 lib/string/memccpy.c 867
95.20.2 lib/string/memchr.c 868
95.20.3 lib/string/memcmp.c 868
95.20.4 lib/string/memcpy.c 868
95.20.5 lib/string/memmove.c 868
95.20.6 lib/string/memset.c 869
95.20.7 lib/string/strcat.c 869
95.20.8 lib/string/strchr.c 869
95.20.9 lib/string/strcmp.c 870
95.20.10 lib/string/strcoll.c 870
95.20.11 lib/string/strcpy.c 870
95.20.12 lib/string/strcspn.c 870
95.20.13 lib/string/strdup.c 871
95.20.14 lib/string/strerror.c 871
95.20.15 lib/string/strlen.c 873
95.20.16 lib/string/strncat.c 873
95.20.17 lib/string/strncmp.c 873
95.20.18 lib/string/strncpy.c 873
95.20.19 lib/string/strpbrk.c 874
95.20.20 lib/string/strrchr.c 874
95.20.21 lib/string/strspn.c 874
95.20.22 lib/string/strstr.c 875
95.20.23 lib/string/strtok.c 875
95.20.24 lib/string/strxfrm.c 877

95.20.1 lib/string/memccpy.c

«

Si veda la sezione 88.77.

```
4500001 #include <string.h>
4500002 //---------------------------------------------------------
4500003 void *
4500004 memccpy (void *restrict dst, const void *restrict org,
4500005          int c, size_t n)
4500006 {
4500007   char *d = (char *) dst;
4500008   char *o = (char *) org;
4500009   size_t i;
4500010   for (i = 0; n > 0 && i < n; i++)
```

```
4500011     {
4500012       d[i] = o[i];
4500013       if (d[i] == (char) c)
4500014         {
4500015           return ((void *) &d[i + 1]);
4500016         }
4500017     }
4500018   return (NULL);
4500019 }
```

95.20.2 lib/string/memchr.c

«

Si veda la sezione 88.78.

```
4510001 #include <string.h>
4510002 //-------------------------------------------------------------
4510003 void *
4510004 memchr (const void *memory, int c, size_t n)
4510005 {
4510006   char *m = (char *) memory;
4510007   size_t i;
4510008   for (i = 0; n > 0 && i < n; i++)
4510009     {
4510010       if (m[i] == (char) c)
4510011         {
4510012           return (void *) (m + i);
4510013         }
4510014     }
4510015   return NULL;
4510016 }
```

95.20.3 lib/string/memcmp.c

«

Si veda la sezione 88.79.

```
4520001 #include <string.h>
4520002 //-------------------------------------------------------------
4520003 int
4520004 memcmp (const void *memory1, const void *memory2, size_t n)
4520005 {
4520006   char *a = (char *) memory1;
4520007   char *b = (char *) memory2;
4520008   size_t i;
4520009   for (i = 0; n > 0 && i < n; i++)
4520010     {
4520011       if (a[i] > b[i])
4520012         {
4520013           return 1;
4520014         }
4520015       else if (a[i] < b[i])
4520016         {
4520017           return -1;
4520018         }
4520019     }
4520020   return 0;
4520021 }
```

95.20.4 lib/string/memcpy.c

«

Si veda la sezione 88.80.

```
4530001 #include <string.h>
4530002 //-------------------------------------------------------------
4530003 void *
4530004 memcpy (void *restrict dst, const void *restrict org,
4530005         size_t n)
4530006 {
4530007   char *d = (char *) dst;
4530008   char *o = (char *) org;
4530009   size_t i;
4530010   for (i = 0; n > 0 && i < n; i++)
4530011     {
4530012       d[i] = o[i];
4530013     }
4530014   return dst;
4530015 }
```

95.20.5 lib/string/memmove.c

«

Si veda la sezione 88.81.

```
4540001 #include <string.h>
4540002 //-------------------------------------------------------------
4540003 void *
4540004 memmove (void *dst, const void *org, size_t n)
4540005 {
```



```
4540006   char *d = (char *) dst;
4540007   char *o = (char *) org;
4540008   size_t i;
4540009   //
4540010   // Depending on the memory start locations, copy may
4540011   // be direct or
4540012   // reverse, to avoid overwriting before the
4540013   // relocation is done.
4540014   //
4540015   if (d < o)
4540016     {
4540017       for (i = 0; i < n; i++)
4540018         {
4540019           d[i] = o[i];
4540020         }
4540021     }
4540022   else if (d == o)
4540023     {
4540024       //
4540025       // Memory locations are already the same.
4540026       //
4540027       ;
4540028     }
4540029   else
4540030     {
4540031       for (i = n - 1; i >= 0; i--)
4540032         {
4540033           d[i] = o[i];
4540034         }
4540035     }
4540036   return dst;
4540037 }
```

95.20.6 lib/string/memset.c

«

Si veda la sezione 88.82.

```
4550001 #include <string.h>
4550002 //-------------------------------------------------------------
4550003 void *
4550004 memset (void *memory, int c, size_t n)
4550005 {
4550006   char *m = (char *) memory;
4550007   size_t i;
4550008   for (i = 0; n > 0 && i < n; i++)
4550009     {
4550010       m[i] = (char) c;
4550011     }
4550012   return memory;
4550013 }
```

95.20.7 lib/string/strcat.c

«

Si veda la sezione 88.113.

```
4560001 #include <string.h>
4560002 //-------------------------------------------------------------
4560003 char *
4560004 strcat (char *restrict dst, const char *restrict org)
4560005 {
4560006   size_t i;
4560007   size_t j;
4560008   for (i = 0; dst[i] != 0; i++)
4560009     {
4560010       ; // Just look for the null character.
4560011     }
4560012   for (j = 0; org[j] != 0; i++, j++)
4560013     {
4560014       dst[i] = org[j];
4560015     }
4560016   dst[i] = 0;
4560017   return dst;
4560018 }
```

95.20.8 lib/string/strchr.c

«

Si veda la sezione 88.114.

```
4570001 #include <string.h>
4570002 //-------------------------------------------------------------
4570003 char *
4570004 strchr (const char *string, int c)
4570005 {
4570006   size_t i;
4570007   for (i = 0;; i++)
4570008     {
4570009       if (string[i] == (char) c)
```

```
4570010         {
4570011           return (char *) (string + i);
4570012         }
4570013       else if (string[i] == 0)
4570014         {
4570015           return NULL;
4570016         }
4570017     }
4570018 }
```

95.20.9 lib/string/strcmp.c

«

Si veda la sezione 88.115.

```
4580001 #include <string.h>
4580002 //-------------------------------------------------------
4580003 int
4580004 strcmp (const char *string1, const char *string2)
4580005 {
4580006   char *a = (char *) string1;
4580007   char *b = (char *) string2;
4580008   size_t i;
4580009   for (i = 0;; i++)
4580010     {
4580011       if (a[i] > b[i])
4580012         {
4580013           return 1;
4580014         }
4580015       else if (a[i] < b[i])
4580016         {
4580017           return -1;
4580018         }
4580019       else if (a[i] == 0 && b[i] == 0)
4580020         {
4580021           return 0;
4580022         }
4580023     }
4580024 }
```

95.20.10 lib/string/strcoll.c

«

Si veda la sezione 88.115.

```
4590001 #include <string.h>
4590002 //-------------------------------------------------------
4590003 int
4590004 strcoll (const char *string1, const char *string2)
4590005 {
4590006   return (strcmp (string1, string2));
4590007 }
```

95.20.11 lib/string/strcpy.c

«

Si veda la sezione 88.117.

```
4600001 #include <string.h>
4600002 //-------------------------------------------------------
4600003 char *
4600004 strcpy (char *restrict dst, const char *restrict org)
4600005 {
4600006   size_t i;
4600007   for (i = 0; org[i] != 0; i++)
4600008     {
4600009       dst[i] = org[i];
4600010     }
4600011   dst[i] = 0;
4600012   return dst;
4600013 }
```

95.20.12 lib/string/strcspn.c

«

Si veda la sezione 88.127.

```
4610001 #include <string.h>
4610002 //-------------------------------------------------------
4610003 size_t
4610004 strcspn (const char *string, const char *reject)
4610005 {
4610006   size_t i;
4610007   size_t j;
4610008   int found;
4610009   for (i = 0; string[i] != 0; i++)
4610010     {
4610011       for (j = 0, found = 0; reject[j] != 0 || found; j++)
4610012         {
4610013           if (string[i] == reject[j])
```



```
4610014             {
4610015               found = 1;
4610016               break;
4610017             }
4610018         }
4610019       if (found)
4610020         {
4610021           break;
4610022         }
4610023     }
4610024   return i;
4610025 }
```

95.20.13 lib/string/strdup.c

«

Si veda la sezione 88.119.

```
4620001 #include <string.h>
4620002 #include <stdlib.h>
4620003 #include <errno.h>
4620004 //-------------------------------------------------------
4620005 char *
4620006 strdup (const char *string)
4620007 {
4620008   size_t size;
4620009   char *copy;
4620010   //
4620011   // Get string size: must be added 1, to count the
4620012   // termination null
4620013   // character.
4620014   //
4620015   size = strlen (string) + 1;
4620016   //
4620017   copy = malloc (size);
4620018   //
4620019   if (copy == NULL)
4620020     {
4620021       errset (ENOMEM);  // Not enough memory.
4620022       return (NULL);
4620023     }
4620024   //
4620025   strcpy (copy, string);
4620026   //
4620027   return (copy);
4620028 }
```

95.20.14 lib/string/strerror.c

«

Si veda la sezione 88.120.

```
4630001 #include <string.h>
4630002 #include <errno.h>
4630003 //-------------------------------------------------------
4630004 #define ERROR_MAX 120
4630005 //-------------------------------------------------------
4630006 char *
4630007 strerror (int errnum)
4630008 {
4630009   static char *err[ERROR_MAX];
4630010   //
4630011   err[0] = "No error";
4630012   err[E2BIG] = TEXT_E2BIG;
4630013   err[EACCES] = TEXT_EACCES;
4630014   err[EADDRINUSE] = TEXT_EADDRINUSE;
4630015   err[EADDRNOTAVAIL] = TEXT_EADDRNOTAVAIL;
4630016   err[EAFNOSUPPORT] = TEXT_EAFNOSUPPORT;
4630017   err[EAGAIN] = TEXT_EAGAIN;
4630018   err[EALREADY] = TEXT_EALREADY;
4630019   err[EBADF] = TEXT_EBADF;
4630020   err[EBADMSG] = TEXT_EBADMSG;
4630021   err[EBUSY] = TEXT_EBUSY;
4630022   err[ECANCELED] = TEXT_ECANCELED;
4630023   err[ECHILD] = TEXT_ECHILD;
4630024   err[ECONNABORTED] = TEXT_ECONNABORTED;
4630025   err[ECONNREFUSED] = TEXT_ECONNREFUSED;
4630026   err[ECONNRESET] = TEXT_ECONNRESET;
4630027   err[EDEADLK] = TEXT_EDEADLK;
4630028   err[EDESTADDRREQ] = TEXT_EDESTADDRREQ;
4630029   err[EDOM] = TEXT_EDOM;
4630030   err[EDQUOT] = TEXT_EDQUOT;
4630031   err[EEXIST] = TEXT_EEXIST;
4630032   err[EFAULT] = TEXT_EFAULT;
4630033   err[EFBIG] = TEXT_EFBIG;
4630034   err[EHOSTUNREACH] = TEXT_EHOSTUNREACH;
4630035   err[EIDRM] = TEXT_EIDRM;
4630036   err[EILSEQ] = TEXT_EILSEQ;
4630037   err[EINPROGRESS] = TEXT_EINPROGRESS;
```

```
4630038   err[EINTR] = TEXT_EINTR;
4630039   err[EINVAL] = TEXT_EINVAL;
4630040   err[EIO] = TEXT_EIO;
4630041   err[EISCONN] = TEXT_EISCONN;
4630042   err[EISDIR] = TEXT_EISDIR;
4630043   err[ELOOP] = TEXT_ELOOP;
4630044   err[EMFILE] = TEXT_EMFILE;
4630045   err[EMLINK] = TEXT_EMLINK;
4630046   err[EMSGSIZE] = TEXT_EMSGSIZE;
4630047   err[EMULTIHOP] = TEXT_EMULTIHOP;
4630048   err[ENAMETOOLONG] = TEXT_ENAMETOOLONG;
4630049   err[ENETDOWN] = TEXT_ENETDOWN;
4630050   err[ENETRESET] = TEXT_ENETRESET;
4630051   err[ENETUNREACH] = TEXT_ENETUNREACH;
4630052   err[ENFILE] = TEXT_ENFILE;
4630053   err[ENOBUFS] = TEXT_ENOBUFS;
4630054   err[ENODATA] = TEXT_ENODATA;
4630055   err[ENODEV] = TEXT_ENODEV;
4630056   err[ENOENT] = TEXT_ENOENT;
4630057   err[ENOEXEC] = TEXT_ENOEXEC;
4630058   err[ENOLCK] = TEXT_ENOLCK;
4630059   err[ENOLINK] = TEXT_ENOLINK;
4630060   err[ENOMEM] = TEXT_ENOMEM;
4630061   err[ENOMSG] = TEXT_ENOMSG;
4630062   err[ENOPROTOOPT] = TEXT_ENOPROTOOPT;
4630063   err[ENOSPC] = TEXT_ENOSPC;
4630064   err[ENOSR] = TEXT_ENOSR;
4630065   err[ENOSTR] = TEXT_ENOSTR;
4630066   err[ENOSYS] = TEXT_ENOSYS;
4630067   err[ENOTCONN] = TEXT_ENOTCONN;
4630068   err[ENOTDIR] = TEXT_ENOTDIR;
4630069   err[ENOTEMPTY] = TEXT_ENOTEMPTY;
4630070   err[ENOTSOCK] = TEXT_ENOTSOCK;
4630071   err[ENOTSUP] = TEXT_ENOTSUP;
4630072   err[ENOTTY] = TEXT_ENOTTY;
4630073   err[ENXIO] = TEXT_ENXIO;
4630074   err[EOPNOTSUPP] = TEXT_EOPNOTSUPP;
4630075   err[EOVERFLOW] = TEXT_EOVERFLOW;
4630076   err[EPERM] = TEXT_EPERM;
4630077   err[EPIPE] = TEXT_EPIPE;
4630078   err[EPROTO] = TEXT_EPROTO;
4630079   err[EPROTONOSUPPORT] = TEXT_EPROTONOSUPPORT;
4630080   err[EPROTOTYPE] = TEXT_EPROTOTYPE;
4630081   err[ERANGE] = TEXT_ERANGE;
4630082   err[EROFS] = TEXT_EROFS;
4630083   err[ESPIPE] = TEXT_ESPIPE;
4630084   err[ESRCH] = TEXT_ESRCH;
4630085   err[ESTALE] = TEXT_ESTALE;
4630086   err[ETIME] = TEXT_ETIME;
4630087   err[ETIMEDOUT] = TEXT_ETIMEDOUT;
4630088   err[ETXTBSY] = TEXT_ETXTBSY;
4630089   err[EWOULDBLOCK] = TEXT_EWOULDBLOCK;
4630090   err[EXDEV] = TEXT_EXDEV;
4630091   err[E_NO_MEDIUM] = TEXT_E_NO_MEDIUM;
4630092   err[E_MEDIUM] = TEXT_E_MEDIUM;
4630093   err[E_FILE_TYPE] = TEXT_E_FILE_TYPE;
4630094   err[E_ROOT_INODE_NOT_CACHED] =
4630095     TEXT_E_ROOT_INODE_NOT_CACHED;
4630096   err[E_CANNOT_READ_SUPERBLOCK] =
4630097     TEXT_E_CANNOT_READ_SUPERBLOCK;
4630098   err[E_MAP_INODE_TOO_BIG] = TEXT_E_MAP_INODE_TOO_BIG;
4630099   err[E_MAP_ZONE_TOO_BIG] = TEXT_E_MAP_ZONE_TOO_BIG;
4630100   err[E_DATA_ZONE_TOO_BIG] = TEXT_E_DATA_ZONE_TOO_BIG;
4630101   err[E_CANNOT_FIND_ROOT_DEVICE] =
4630102     TEXT_E_CANNOT_FIND_ROOT_DEVICE;
4630103   err[E_CANNOT_FIND_ROOT_INODE] =
4630104     TEXT_E_CANNOT_FIND_ROOT_INODE;
4630105   err[E_FILE_TYPE_UNSUPPORTED] =
4630106     TEXT_E_FILE_TYPE_UNSUPPORTED;
4630107   err[E_ENV_TOO_BIG] = TEXT_E_ENV_TOO_BIG;
4630108   err[E_LIMIT] = TEXT_E_LIMIT;
4630109   err[E_NOT_MOUNTED] = TEXT_E_NOT_MOUNTED;
4630110   err[E_NOT_IMPLEMENTED] = TEXT_E_NOT_IMPLEMENTED;
4630111   err[E_HARDWARE_FAULT] = TEXT_E_HARDWARE_FAULT;
4630112   err[E_DRIVER_FAULT] = TEXT_E_DRIVER_FAULT;
4630113   err[E_PIPE_FULL] = TEXT_E_PIPE_FULL;
4630114   err[E_PIPE_EMPTY] = TEXT_E_PIPE_EMPTY;
4630115   err[E_PART_TYPE_NOT_MINIX] = TEXT_E_PART_TYPE_NOT_MINIX;
4630116   err[E_FS_TYPE_NOT_SUPPORTED] =
4630117     TEXT_E_FS_TYPE_NOT_SUPPORTED;
4630118   err[E_PDU_TOO_BIG] = TEXT_E_PDU_TOO_BIG;
4630119   err[E_ARP_MISSING] = TEXT_E_ARP_MISSING;
4630120   //
4630121   if (errnum >= ERROR_MAX || errnum < 0)
4630122     {
4630123       return ("Unknown error");
4630124     }
```



```
4630125   //
4630126   return (err[errnum]);
4630127 }
```

95.20.15 lib/string/strlen.c

Si veda la sezione 88.121.

```
4640001 #include <string.h>
4640002 //---------------------------------------------------------
4640003 size_t
4640004 strlen (const char *string)
4640005 {
4640006   size_t i;
4640007   for (i = 0; string[i] != 0; i++)
4640008     {
4640009       ; // Just count.
4640010     }
4640011   return i;
4640012 }
```

95.20.16 lib/string/strncat.c

Si veda la sezione 88.113.

```
4650001 #include <string.h>
4650002 //---------------------------------------------------------
4650003 char *
4650004 strncat (char *restrict dst, const char *restrict org,
4650005          size_t n)
4650006 {
4650007   size_t i;
4650008   size_t j;
4650009   for (i = 0; n > 0 && dst[i] != 0; i++)
4650010     {
4650011       ; // Just seek the null character.
4650012     }
4650013   for (j = 0; n > 0 && j < n && org[j] != 0; i++, j++)
4650014     {
4650015       dst[i] = org[j];
4650016     }
4650017   dst[i] = 0;
4650018   return dst;
4650019 }
```

95.20.17 lib/string/strncmp.c

Si veda la sezione 88.115.

```
4660001 #include <string.h>
4660002 //---------------------------------------------------------
4660003 int
4660004 strncmp (const char *string1, const char *string2, size_t n)
4660005 {
4660006   size_t i;
4660007   for (i = 0; i < n; i++)
4660008     {
4660009       if (string1[i] > string2[i])
4660010         {
4660011           return 1;
4660012         }
4660013       else if (string1[i] < string2[i])
4660014         {
4660015           return -1;
4660016         }
4660017       else if (string1[i] == 0 && string2[i] == 0)
4660018         {
4660019           return 0;
4660020         }
4660021     }
4660022   return 0;
4660023 }
```

95.20.18 lib/string/strncpy.c

Si veda la sezione 88.117.

```
4670001 #include <string.h>
4670002 //---------------------------------------------------------
4670003 char *
4670004 strncpy (char *restrict dst, const char *restrict org,
4670005          size_t n)
4670006 {
4670007   size_t i;
4670008   for (i = 0; n > 0 && i < n && org[i] != 0; i++)
4670009     {
```

```
4670010         dst[i] = org[i];
4670011       }
4670012     for (; n > 0 && i < n; i++)
4670013       {
4670014         dst[i] = 0;
4670015       }
4670016     return dst;
4670017   }
```

95.20.19 lib/string/strpbrk.c

Si veda la sezione 88.125.

```
4680001 #include <string.h>
4680002 //-------------------------------------------------------------
4680003 char *
4680004 strpbrk (const char *string, const char *accept)
4680005 {
4680006   //
4680007   // The first parameter not 'const char *' because
4680008   // otherwise
4680009   // the return value should be 'const char *' too!
4680010   //
4680011   size_t i;
4680012   size_t j;
4680013   //
4680014   for (i = 0; string[i] != 0; i++)
4680015     {
4680016       for (j = 0; accept[j] != 0; j++)
4680017         {
4680018           if (string[i] == accept[j])
4680019             {
4680020               return (char *) (string + i);
4680021             }
4680022         }
4680023     }
4680024   return NULL;
4680025 }
```

95.20.20 lib/string/strrchr.c

Si veda la sezione 88.114.

```
4690001 #include <string.h>
4690002 //-------------------------------------------------------------
4690003 char *
4690004 strrchr (const char *string, int c)
4690005 {
4690006   int i;
4690007   for (i = strlen (string); i >= 0; i--)
4690008     {
4690009       if (string[i] == (char) c)
4690010         {
4690011           break;
4690012         }
4690013     }
4690014   if (i < 0)
4690015     {
4690016       return NULL;
4690017     }
4690018   else
4690019     {
4690020       return (char *) (string + i);
4690021     }
4690022 }
```

95.20.21 lib/string/strspn.c

Si veda la sezione 88.127.

```
4700001 #include <string.h>
4700002 //-------------------------------------------------------------
4700003 size_t
4700004 strspn (const char *string, const char *accept)
4700005 {
4700006   size_t i;
4700007   size_t j;
4700008   int found;
4700009   for (i = 0; string[i] != 0; i++)
4700010     {
4700011       for (j = 0, found = 0; accept[j] != 0; j++)
4700012         {
4700013           if (string[i] == accept[j])
4700014             {
4700015               found = 1;
4700016               break;
4700017             }
```



```
4700018         }
4700019       if (!found)
4700020         {
4700021           break;
4700022         }
4700023     }
4700024   return i;
4700025 }
```

95.20.22 lib/string/strstr.c

Si veda la sezione 88.128.

```
4710001 #include <string.h>
4710002 //-------------------------------------------------------------
4710003 char *
4710004 strstr (const char *string, const char *substring)
4710005 {
4710006   size_t i;
4710007   size_t j;
4710008   size_t k;
4710009   int found;
4710010   if (substring[0] == 0)
4710011     {
4710012       return (char *) string;
4710013     }
4710014   for (i = 0, j = 0, found = 0; string[i] != 0; i++)
4710015     {
4710016       if (string[i] == substring[0])
4710017         {
4710018           for (k = i, j = 0;
4710019                string[k] == substring[j] &&
4710020                string[k] != 0 &&
4710021                substring[j] != 0; j++, k++)
4710022             {
4710023               ;
4710024             }
4710025           if (substring[j] == 0)
4710026             {
4710027               found = 1;
4710028             }
4710029         }
4710030       if (found)
4710031         {
4710032           return (char *) (string + i);
4710033         }
4710034     }
4710035   return NULL;
4710036 }
```

95.20.23 lib/string/strtok.c

Si veda la sezione 88.129.

```
4720001 #include <string.h>
4720002 //-------------------------------------------------------------
4720003 char *
4720004 strtok (char *restrict string, const char *restrict delim)
4720005 {
4720006   static char *next = NULL;
4720007   size_t i = 0;
4720008   size_t j;
4720009   int found_token;
4720010   int found_delim;
4720011   //
4720012   // If the string received a the first parameter is a
4720013   // null pointer,
4720014   // the static pointer is used. But if it is already
4720015   // NULL,
4720016   // the scan cannot start.
4720017   //
4720018   if (string == NULL)
4720019     {
4720020       if (next == NULL)
4720021         {
4720022           return NULL;
4720023         }
4720024       else
4720025         {
4720026           string = next;
4720027         }
4720028     }
4720029   //
4720030   // If the string received as the first parameter is
4720031   // empty, the scan
4720032   // cannot start.
4720033   //
```

```
4720034   if (string[0] == 0)
4720035     {
4720036       next = NULL;
4720037       return NULL;
4720038     }
4720039   else
4720040     {
4720041       if (delim[0] == 0)
4720042         {
4720043           return string;
4720044         }
4720045     }
4720046   //
4720047   // Find the next token.
4720048   //
4720049   for (i = 0, found_token = 0, j = 0;
4720050        string[i] != 0 && (!found_token); i++)
4720051     {
4720052       //
4720053       // Look inside delimiters.
4720054       //
4720055       for (j = 0, found_delim = 0; delim[j] != 0; j++)
4720056         {
4720057           if (string[i] == delim[j])
4720058             {
4720059               found_delim = 1;
4720060             }
4720061         }
4720062       //
4720063       // If current character inside the string is not
4720064       // a delimiter,
4720065       // it is the start of a new token.
4720066       //
4720067       if (!found_delim)
4720068         {
4720069           found_token = 1;
4720070           break;
4720071         }
4720072     }
4720073   //
4720074   // If a token was found, the pointer is updated.
4720075   // If otherwise the token is not found, this means
4720076   // that
4720077   // there are no more.
4720078   //
4720079   if (found_token)
4720080     {
4720081       string += i;
4720082     }
4720083   else
4720084     {
4720085       next = NULL;
4720086       return NULL;
4720087     }
4720088   //
4720089   // Find the end of the token.
4720090   //
4720091   for (i = 0, found_delim = 0; string[i] != 0; i++)
4720092     {
4720093       for (j = 0; delim[j] != 0; j++)
4720094         {
4720095           if (string[i] == delim[j])
4720096             {
4720097               found_delim = 1;
4720098               break;
4720099             }
4720100         }
4720101       if (found_delim)
4720102         {
4720103           break;
4720104         }
4720105     }
4720106   //
4720107   // If a delimiter was found, the corresponding
4720108   // character must be
4720109   // reset to zero. If otherwise the string is
4720110   // terminated, the
4720111   // scan is terminated.
4720112   //
4720113   if (found_delim)
4720114     {
4720115       string[i] = 0;
4720116       next = &string[i + 1];
4720117     }
4720118   else
4720119     {
4720120       next = NULL;
```



```
4720121     }
4720122   //
4720123   // At this point, the current string represent the
4720124   // token found.
4720125   //
4720126   return string;
4720127 }
```

95.20.24 lib/string/strxfrm.c

Si veda la sezione 88.132.

```
4730001 #include <string.h>
4730002 //---------------------------------------------------------
4730003 size_t
4730004 strxfrm (char *restrict dst, const char *restrict org,
4730005          size_t n)
4730006 {
4730007   size_t i;
4730008   if (n == 0 && dst == NULL)
4730009     {
4730010       return strlen (org);
4730011     }
4730012   else
4730013     {
4730014       for (i = 0; i < n; i++)
4730015         {
4730016           dst[i] = org[i];
4730017           if (org[i] == 0)
4730018             {
4730019               break;
4730020             }
4730021         }
4730022       return i;
4730023     }
4730024 }
```

95.21 os32: «lib/sys/os32.h»

Si veda la sezione 91.3.

```
4740001 #ifndef _SYS_OS32_H
4740002 #define _SYS_OS32_H      1
4740003 //---------------------------------------------------------
4740004 // This file contains all the declarations that don't
4740005 // have a better place inside standard headers files.
4740006 // Even declarations related to device numbers and
4740007 // system calls is contained here.
4740008 //---------------------------------------------------------
4740009 #include <sys/types.h>
4740010 #include <sys/stat.h>
4740011 #include <sys/socket.h>
4740012 #include <arpa/inet.h>
4740013 #include <netinet/in.h>
4740014 #include <stdint.h>
4740015 #include <signal.h>
4740016 #include <limits.h>
4740017 #include <stdio.h>
4740018 #include <stddef.h>
4740019 #include <restrict.h>
4740020 #include <stdarg.h>
4740021 #include <termios.h>
4740022 //---------------------------------------------------------
4740023 typedef uint16_t h_port_t;      // Port number in host
4740024                                 // byte order.
4740025 typedef uint32_t h_addr_t;      // IPv4 address in
4740026                                 // host byte order.
4740027 //---------------------------------------------------------
4740028 // Please remember that system calls should never be
4740029 // used (called) inside the kernel code, because system
4740030 // calls cannot be nested for the os32 simple
4740031 // architecture!
4740032 // If a particular function is necessary inside the
4740033 // kernel, that usually is made by a system call, an
4740034 // appropriate k_...() function must be
4740035 // made, to avoid the problem.
4740036 //---------------------------------------------------------
4740037 // Device numbers.
4740038 //---------------------------------------------------------
4740039 #define DEV_UNDEFINED_MAJOR     ((dev_t) 0x00)
4740040 #define DEV_UNDEFINED           ((dev_t) 0x0000)
4740041 #define DEV_MEM_MAJOR           ((dev_t) 0x01)
4740042 #define DEV_MEM                 ((dev_t) 0x0101)
4740043 #define DEV_NULL                ((dev_t) 0x0102)
4740044 #define DEV_PORT                ((dev_t) 0x0103)
4740045 #define DEV_ZERO                ((dev_t) 0x0104)
4740046 #define DEV_TTY_MAJOR           ((dev_t) 0x02)
```

```
#define DEV_TTY                 ((dev_t) 0x0200)
//
#define DEV_KMEM_MAJOR          ((dev_t) 0x04)
#define DEV_KMEM_PS             ((dev_t) 0x0401)
#define DEV_KMEM_MMP            ((dev_t) 0x0402)
#define DEV_KMEM_SB             ((dev_t) 0x0403)
#define DEV_KMEM_INODE          ((dev_t) 0x0404)
#define DEV_KMEM_FILE           ((dev_t) 0x0405)
#define DEV_KMEM_ARP            ((dev_t) 0x0406)
#define DEV_KMEM_NET            ((dev_t) 0x0407)
#define DEV_KMEM_ROUTE          ((dev_t) 0x0408)
//
#define DEV_CONSOLE_MAJOR       ((dev_t) 0x05)
#define DEV_CONSOLE             ((dev_t) 0x05FF)
#define DEV_CONSOLE0            ((dev_t) 0x0500)
#define DEV_CONSOLE1            ((dev_t) 0x0501)
#define DEV_CONSOLE2            ((dev_t) 0x0502)
#define DEV_CONSOLE3            ((dev_t) 0x0503)
#define DEV_CONSOLE4            ((dev_t) 0x0504)
//
#define DEV_DM_MAJOR            ((dev_t) 0x08)
#define DEV_DM00                ((dev_t) 0x0800)
#define DEV_DM01                ((dev_t) 0x0801)
#define DEV_DM02                ((dev_t) 0x0802)
#define DEV_DM03                ((dev_t) 0x0803)
#define DEV_DM04                ((dev_t) 0x0804)
#define DEV_DM10                ((dev_t) 0x0810)
#define DEV_DM11                ((dev_t) 0x0811)
#define DEV_DM12                ((dev_t) 0x0812)
#define DEV_DM13                ((dev_t) 0x0813)
#define DEV_DM14                ((dev_t) 0x0814)
#define DEV_DM20                ((dev_t) 0x0820)
#define DEV_DM21                ((dev_t) 0x0821)
#define DEV_DM22                ((dev_t) 0x0822)
#define DEV_DM23                ((dev_t) 0x0823)
#define DEV_DM24                ((dev_t) 0x0824)
#define DEV_DM30                ((dev_t) 0x0830)
#define DEV_DM31                ((dev_t) 0x0831)
#define DEV_DM32                ((dev_t) 0x0832)
#define DEV_DM33                ((dev_t) 0x0833)
#define DEV_DM34                ((dev_t) 0x0834)
//
//---------------------------------------------------------
#define min(a, b) (a < b ? a : b)
#define max(a, b) (a > b ? a : b)
#define sizeof_array(x) (sizeof(x) / sizeof((x)[0]))
#define sizeof_field(t, f) (sizeof(((t*)0)->f))
//---------------------------------------------------------
#define INPUT_LINE_HIDDEN 0
#define INPUT_LINE_ECHO   1
//---------------------------------------------------------
#define MOUNT_DEFAULT     0     // Default mount
                                // options.
#define MOUNT_RO          1     // Read only mount
                                // option.
//---------------------------------------------------------
#define SYS_0                    0      // Nothing to
                                        // do.
#define SYS_CHDIR                1
#define SYS_CHMOD                2
#define SYS_CLOCK                3
#define SYS_CLOSE                4
#define SYS_EXEC                 5
#define SYS_EXIT                 6      // [1] see
                                        // below.
#define SYS_FCHMOD               7
#define SYS_FORK                 8
#define SYS_FSTAT                9
#define SYS_KILL                10
#define SYS_LSEEK               11
#define SYS_MKDIR               12
#define SYS_MKNOD               13
#define SYS_MOUNT               14
#define SYS_OPEN                15
#define SYS_PGRP                16
#define SYS_READ                17
#define SYS_SETEUID             18
#define SYS_SETUID              19
#define SYS_SIGNAL              20
#define SYS_SLEEP               21
#define SYS_STAT                22
#define SYS_TIME                23
#define SYS_UAREA               24
#define SYS_UMASK               25
#define SYS_UMOUNT              26
#define SYS_WAIT                27
#define SYS_WRITE               28
```



```
#define SYS_ZPCHAR              29      // [2]
#define SYS_ZPSTRING            30      // [2]
#define SYS_CHOWN               31
#define SYS_DUP                 33
#define SYS_DUP2                34
#define SYS_LINK                35
#define SYS_UNLINK              36
#define SYS_FCNTL               37
#define SYS_STIME               38
#define SYS_FCHOWN              39
#define SYS_BRK                 40
#define SYS_SBRK                41
#define SYS_PIPE                42
#define SYS_TCGETATTR           43
#define SYS_TCSETATTR           44
#define SYS_SETEGID             45
#define SYS_SETGID              46
#define SYS_SETJMP              47
#define SYS_LONGJMP             48
#define SYS_RECVFROM            49
#define SYS_SOCKET              50
#define SYS_CONNECT             51
#define SYS_SEND                52
#define SYS_IPCONFIG            53
#define SYS_ROUTEADD            54
#define SYS_ROUTEDEL            55
#define SYS_BIND                56
#define SYS_LISTEN              57
#define SYS_ACCEPT              58
//
// [1] The files 'crt0...' need to know the value used
//     for the exit system call. If this value is
//     modified, all the file 'crt0...' have also to be
//     modified the same way.
//
// [2] These system calls were developed at the
//     beginning, when no standard I/O was available.
//     They are to be considered as a last resort for
//     debugging purposes.
//
//---------------------------------------------------------
// The following values must be: 1, 2, 4, 8, 16, 32,...
// so that can be 'OR' combined.
//
#define WAKEUP_EVENT_SIGNAL       0x0001
#define WAKEUP_EVENT_TIMER        0x0002
#define WAKEUP_EVENT_DEV_READ     0x0004
#define WAKEUP_EVENT_DEV_WRITE    0x0008
#define WAKEUP_EVENT_PIPE_READ    0x0010
#define WAKEUP_EVENT_PIPE_WRITE   0x0020
#define WAKEUP_EVENT_SOCK_READ    0x0040
#define WAKEUP_EVENT_SOCK_WRITE   0x0080
//---------------------------------------------------------
typedef struct
{
  int sfdn;
  struct sockaddr addr;
  socklen_t addrlen;
  int fl_flags;
  int ret;
  int errno;
  int errln;
  char errfn[PATH_MAX];
} sysmsg_accept_t;
//---------------------------------------------------------
typedef struct
{
  int sfdn;
  struct sockaddr addr;
  socklen_t addrlen;
  int ret;
  int errno;
  int errln;
  char errfn[PATH_MAX];
} sysmsg_bind_t;
//---------------------------------------------------------
typedef struct
{
  void *address;
  int ret;
  int errno;
  int errln;
  char errfn[PATH_MAX];
} sysmsg_brk_t;
//---------------------------------------------------------
typedef struct
{
```

```
4740221   const char *path;
4740222   int ret;
4740223   int errno;
4740224   int errln;
4740225   char errfn[PATH_MAX];
4740226 } sysmsg_chdir_t;
4740227 //-------------------------------------------------------
4740228 typedef struct
4740229 {
4740230   const char *path;
4740231   mode_t mode;
4740232   int ret;
4740233   int errno;
4740234   int errln;
4740235   char errfn[PATH_MAX];
4740236 } sysmsg_chmod_t;
4740237 //-------------------------------------------------------
4740238 typedef struct
4740239 {
4740240   const char *path;
4740241   uid_t uid;
4740242   uid_t gid;
4740243   int ret;
4740244   int errno;
4740245   int errln;
4740246   char errfn[PATH_MAX];
4740247 } sysmsg_chown_t;
4740248 //-------------------------------------------------------
4740249 typedef struct
4740250 {
4740251   clock_t ret;
4740252 } sysmsg_clock_t;
4740253 //-------------------------------------------------------
4740254 typedef struct
4740255 {
4740256   int fdn;
4740257   int ret;
4740258   int errno;
4740259   int errln;
4740260   char errfn[PATH_MAX];
4740261 } sysmsg_close_t;
4740262 //-------------------------------------------------------
4740263 typedef struct
4740264 {
4740265   int sfdn;
4740266   struct sockaddr addr;
4740267   socklen_t addrlen;
4740268   int ret;
4740269   int errno;
4740270   int errln;
4740271   char errfn[PATH_MAX];
4740272 } sysmsg_connect_t;
4740273 //-------------------------------------------------------
4740274 typedef struct
4740275 {
4740276   int fdn_old;
4740277   int ret;
4740278   int errno;
4740279   int errln;
4740280   char errfn[PATH_MAX];
4740281 } sysmsg_dup_t;
4740282 //-------------------------------------------------------
4740283 typedef struct
4740284 {
4740285   int fdn_old;
4740286   int fdn_new;
4740287   int ret;
4740288   int errno;
4740289   int errln;
4740290   char errfn[PATH_MAX];
4740291 } sysmsg_dup2_t;
4740292 //-------------------------------------------------------
4740293 typedef struct
4740294 {
4740295   const char *path;
4740296   int argc;
4740297   int envc;
4740298   char arg_data[ARG_MAX / 2];
4740299   char env_data[ARG_MAX / 2];
4740300   uid_t uid;
4740301   uid_t euid;
4740302   int ret;
4740303   int errno;
4740304   int errln;
4740305   char errfn[PATH_MAX];
4740306 } sysmsg_exec_t;
4740307 //-------------------------------------------------------
```



```
4740308 typedef struct
4740309 {
4740310   int status;
4740311 } sysmsg_exit_t;
4740312 //-------------------------------------------------------
4740313 typedef struct
4740314 {
4740315   int fdn;
4740316   mode_t mode;
4740317   int ret;
4740318   int errno;
4740319   int errln;
4740320   char errfn[PATH_MAX];
4740321 } sysmsg_fchmod_t;
4740322 //-------------------------------------------------------
4740323 typedef struct
4740324 {
4740325   int fdn;
4740326   uid_t uid;
4740327   uid_t gid;
4740328   int ret;
4740329   int errno;
4740330   int errln;
4740331   char errfn[PATH_MAX];
4740332 } sysmsg_fchown_t;
4740333 //-------------------------------------------------------
4740334 typedef struct
4740335 {
4740336   int fdn;
4740337   int cmd;
4740338   int arg;
4740339   int ret;
4740340   int errno;
4740341   int errln;
4740342   char errfn[PATH_MAX];
4740343 } sysmsg_fcntl_t;
4740344 //-------------------------------------------------------
4740345 typedef struct
4740346 {
4740347   pid_t ret;
4740348   int errno;
4740349   int errln;
4740350   char errfn[PATH_MAX];
4740351 } sysmsg_fork_t;
4740352 //-------------------------------------------------------
4740353 typedef struct
4740354 {
4740355   int fdn;
4740356   struct stat stat;
4740357   int ret;
4740358   int errno;
4740359   int errln;
4740360   char errfn[PATH_MAX];
4740361 } sysmsg_fstat_t;
4740362 //-------------------------------------------------------
4740363 typedef struct
4740364 {
4740365   int n;
4740366   in_addr_t address;
4740367   int m;
4740368   int ret;
4740369   int errno;
4740370   int errln;
4740371   char errfn[PATH_MAX];
4740372 } sysmsg_ipconfig_t;
4740373 //-------------------------------------------------------
4740374 typedef struct
4740375 {
4740376   void *env;
4740377   int ret;
4740378   //
4740379   // This structure is intentionally reduced.
4740380   //
4740381 } sysmsg_jmp_t;
4740382 //-------------------------------------------------------
4740383 typedef struct
4740384 {
4740385   pid_t pid;
4740386   int signal;
4740387   int ret;
4740388   int errno;
4740389   int errln;
4740390   char errfn[PATH_MAX];
4740391 } sysmsg_kill_t;
4740392 //-------------------------------------------------------
4740393 typedef struct
4740394 {
```

```
4740395   const char *path_old;
4740396   const char *path_new;
4740397   int ret;
4740398   int errno;
4740399   int errln;
4740400   char errfn[PATH_MAX];
4740401 } sysmsg_link_t;
4740402 //-------------------------------------------------------
4740403 typedef struct
4740404 {
4740405   int sfdn;
4740406   int backlog;
4740407   int ret;
4740408   int errno;
4740409   int errln;
4740410   char errfn[PATH_MAX];
4740411 } sysmsg_listen_t;
4740412 //-------------------------------------------------------
4740413 typedef struct
4740414 {
4740415   int fdn;
4740416   off_t offset;
4740417   int whence;
4740418   int ret;
4740419   int errno;
4740420   int errln;
4740421   char errfn[PATH_MAX];
4740422 } sysmsg_lseek_t;
4740423 //-------------------------------------------------------
4740424 typedef struct
4740425 {
4740426   const char *path;
4740427   mode_t mode;
4740428   int ret;
4740429   int errno;
4740430   int errln;
4740431   char errfn[PATH_MAX];
4740432 } sysmsg_mkdir_t;
4740433 //-------------------------------------------------------
4740434 typedef struct
4740435 {
4740436   const char *path;
4740437   mode_t mode;
4740438   dev_t device;
4740439   int ret;
4740440   int errno;
4740441   int errln;
4740442   char errfn[PATH_MAX];
4740443 } sysmsg_mknod_t;
4740444 //-------------------------------------------------------
4740445 typedef struct
4740446 {
4740447   const char *path_dev;
4740448   const char *path_mnt;
4740449   int options;
4740450   int ret;
4740451   int errno;
4740452   int errln;
4740453   char errfn[PATH_MAX];
4740454 } sysmsg_mount_t;
4740455 //-------------------------------------------------------
4740456 typedef struct
4740457 {
4740458   const char *path;
4740459   int flags;
4740460   mode_t mode;
4740461   int ret;
4740462   int errno;
4740463   int errln;
4740464   char errfn[PATH_MAX];
4740465 } sysmsg_open_t;
4740466 //-------------------------------------------------------
4740467 typedef struct
4740468 {
4740469   int pipefd[2];
4740470   int ret;
4740471   int errno;
4740472   int errln;
4740473   char errfn[PATH_MAX];
4740474 } sysmsg_pipe_t;
4740475 //-------------------------------------------------------
4740476 typedef struct
4740477 {
4740478   int fdn;
4740479   void *buffer;
4740480   size_t count;
4740481   int fl_flags;
```

```
4740482   ssize_t ret;
4740483   int errno;
4740484   int errln;
4740485   char errfn[PATH_MAX];
4740486 } sysmsg_read_t;
4740487 //-------------------------------------------------------
4740488 typedef struct
4740489 {
4740490   int sfdn;
4740491   void *buffer;
4740492   size_t count;
4740493   int flags;
4740494   void *addrfrom;
4740495   void *addrsize;
4740496   int fl_flags;
4740497   ssize_t ret;
4740498   int errno;
4740499   int errln;
4740500   char errfn[PATH_MAX];
4740501 } sysmsg_recvfrom_t;
4740502 //-------------------------------------------------------
4740503 typedef struct
4740504 {
4740505   in_addr_t destination;
4740506   int m;
4740507   in_addr_t router;
4740508   int device;
4740509   int ret;
4740510   int errno;
4740511   int errln;
4740512   char errfn[PATH_MAX];
4740513 } sysmsg_route_t;
4740514 //-------------------------------------------------------
4740515 typedef struct
4740516 {
4740517   intptr_t increment;
4740518   void *ret;
4740519   int errno;
4740520   int errln;
4740521   char errfn[PATH_MAX];
4740522 } sysmsg_sbrk_t;
4740523 //-------------------------------------------------------
4740524 typedef struct
4740525 {
4740526   int sfdn;
4740527   const void *buffer;
4740528   size_t count;
4740529   int flags;
4740530   ssize_t ret;
4740531   int errno;
4740532   int errln;
4740533   char errfn[PATH_MAX];
4740534 } sysmsg_send_t;
4740535 //-------------------------------------------------------
4740536 typedef struct
4740537 {
4740538   gid_t egid;
4740539   int ret;
4740540   int errno;
4740541   int errln;
4740542   char errfn[PATH_MAX];
4740543 } sysmsg_setegid_t;
4740544 //-------------------------------------------------------
4740545 typedef struct
4740546 {
4740547   uid_t euid;
4740548   int ret;
4740549   int errno;
4740550   int errln;
4740551   char errfn[PATH_MAX];
4740552 } sysmsg_seteuid_t;
4740553 //-------------------------------------------------------
4740554 typedef struct
4740555 {
4740556   gid_t gid;
4740557   gid_t egid;
4740558   gid_t sgid;
4740559   int ret;
4740560   int errno;
4740561   int errln;
4740562   char errfn[PATH_MAX];
4740563 } sysmsg_setgid_t;
4740564 //-------------------------------------------------------
4740565 typedef struct
4740566 {
4740567   uid_t uid;
4740568   uid_t euid;
```

```
4740569   uid_t suid;
4740570   int ret;
4740571   int errno;
4740572   int errln;
4740573   char errfn[PATH_MAX];
4740574 } sysmsg_setuid_t;
4740575 //-------------------------------------------------------
4740576 typedef struct
4740577 {
4740578   uintptr_t wrapper;
4740579   sighandler_t handler;
4740580   int signal;
4740581   sighandler_t ret;
4740582   int errno;
4740583   int errln;
4740584   char errfn[PATH_MAX];
4740585 } sysmsg_signal_t;
4740586 //-------------------------------------------------------
4740587 typedef struct
4740588 {
4740589   int family;
4740590   int type;
4740591   int protocol;
4740592   int ret;
4740593   int errno;
4740594   int errln;
4740595   char errfn[PATH_MAX];
4740596 } sysmsg_socket_t;
4740597 //-------------------------------------------------------
4740598 typedef struct
4740599 {
4740600   int events;
4740601   int signal;
4740602   unsigned int seconds;
4740603   time_t ret;
4740604 } sysmsg_sleep_t;
4740605 //-------------------------------------------------------
4740606 typedef struct
4740607 {
4740608   const char *path;
4740609   struct stat stat;
4740610   int ret;
4740611   int errno;
4740612   int errln;
4740613   char errfn[PATH_MAX];
4740614 } sysmsg_stat_t;
4740615 //-------------------------------------------------------
4740616 typedef struct
4740617 {
4740618   time_t ret;
4740619 } sysmsg_time_t;
4740620 //-------------------------------------------------------
4740621 typedef struct
4740622 {
4740623   time_t timer;
4740624   int ret;
4740625 } sysmsg_stime_t;
4740626 //-------------------------------------------------------
4740627 typedef struct
4740628 {
4740629   int fdn;
4740630   int action;
4740631   struct termios *attr;
4740632   int ret;
4740633   int errno;
4740634   int errln;
4740635   char errfn[PATH_MAX];
4740636 } sysmsg_tcattr_t;
4740637 //-------------------------------------------------------
4740638 typedef struct
4740639 {
4740640   uid_t uid;    // Read user ID.
4740641   uid_t euid;   // Effective user ID.
4740642   uid_t suid;   // Saved user ID.
4740643   gid_t gid;    // Read group ID.
4740644   gid_t egid;   // Effective group ID.
4740645   gid_t sgid;   // Saved group ID.
4740646   pid_t pid;    // Process ID.
4740647   pid_t ppid;   // Parent PID.
4740648   pid_t pgrp;   // Process group.
4740649   mode_t umask; // Access permission mask.
4740650   char *path_cwd;
4740651   size_t path_cwd_size; // Max path size.
4740652 } sysmsg_uarea_t;
4740653 //-------------------------------------------------------
4740654 typedef struct
4740655 {
```



```
4740656   mode_t umask;
4740657   mode_t ret;
4740658 } sysmsg_umask_t;
4740659 //-------------------------------------------------------
4740660 typedef struct
4740661 {
4740662   const char *path_mnt;
4740663   int ret;
4740664   int errno;
4740665   int errln;
4740666   char errfn[PATH_MAX];
4740667 } sysmsg_umount_t;
4740668 //-------------------------------------------------------
4740669 typedef struct
4740670 {
4740671   const char *path;
4740672   int ret;
4740673   int errno;
4740674   int errln;
4740675   char errfn[PATH_MAX];
4740676 } sysmsg_unlink_t;
4740677 //-------------------------------------------------------
4740678 typedef struct
4740679 {
4740680   int status;
4740681   pid_t ret;
4740682   int errno;
4740683   int errln;
4740684   char errfn[PATH_MAX];
4740685 } sysmsg_wait_t;
4740686 //-------------------------------------------------------
4740687 typedef struct
4740688 {
4740689   int fdn;
4740690   const void *buffer;
4740691   size_t count;
4740692   ssize_t ret;
4740693   int errno;
4740694   int errln;
4740695   char errfn[PATH_MAX];
4740696 } sysmsg_write_t;
4740697 //-------------------------------------------------------
4740698 typedef struct
4740699 {
4740700   char c;
4740701 } sysmsg_zpchar_t;
4740702 //-------------------------------------------------------
4740703 typedef struct
4740704 {
4740705   char string[BUFSIZ];
4740706 } sysmsg_zpstring_t;
4740707 //-------------------------------------------------------
4740708 void input_line (char *line, char *prompt, size_t size,
4740709                  int type);
4740710 int mount (const char *path_dev, const char *path_mnt,
4740711           int options);
4740712 int namep (const char *name, char *path, size_t size);
4740713 void sys (int syscallnr, void *message, size_t size);
4740714 int umount (const char *path_mnt);
4740715 void z_perror (const char *string);
4740716 int z_printf (const char *restrict format, ...);
4740717 int z_vprintf (const char *restrict format, va_list arg);
4740718 int ipconfig (int n, h_addr_t address, int m);
4740719 int routedel (h_addr_t destination, int m);
4740720 int routeadd (h_addr_t destination, int m,
4740721               h_addr_t router, int device);
4740722 //-------------------------------------------------------
4740723 #endif
```


95.21.1 lib/sys/os32/input_line.c 885
95.21.2 lib/sys/os32/ipconfig.c 887
95.21.3 lib/sys/os32/mount.c 888
95.21.4 lib/sys/os32/namep.c 888
95.21.5 lib/sys/os32/routeadd.c 890
95.21.6 lib/sys/os32/routedel.c 891
95.21.7 lib/sys/os32/sys.s 891
95.21.8 lib/sys/os32/umount.c 891
95.21.9 lib/sys/os32/z_perror.c 892
95.21.10 lib/sys/os32/z_printf.c 892
95.21.11 lib/sys/os32/z_vprintf.c 892

95.21.1 lib/sys/os32/input_line.c

Si veda la sezione 88.68.

```
4750001 #include <sys/os32.h>
4750002 #include <string.h>
4750003 #include <stdio.h>
4750004 #include <errno.h>
4750005 #include <unistd.h>
4750006 //-------------------------------------------------------
4750007 static int terminal_echo (struct termios *orig);
4750008 static int terminal_noecho (struct termios *orig);
4750009 static int terminal_restore (struct termios *orig);
4750010 //-------------------------------------------------------
4750011 void
4750012 input_line (char *line, char *prompt, size_t size, int type)
4750013 {
4750014   void *pstatus;
4750015   int i;
4750016   struct termios attr;
4750017   //
4750018   // Set terminal configuration.
4750019   //
4750020   if (type == INPUT_LINE_HIDDEN)
4750021     {
4750022       terminal_noecho (&attr);
4750023     }
4750024   else
4750025     {
4750026       terminal_echo (&attr);
4750027     }
4750028   //
4750029   if (prompt != NULL || strlen (prompt) > 0)
4750030     {
4750031       printf ("%s", prompt);
4750032     }
4750033   //
4750034   errno = 0;
4750035   pstatus = fgets (line, (int) size, stdin);
4750036   if (pstatus == NULL)
4750037     {
4750038       if (errno)
4750039         {
4750040           perror (NULL);
4750041         }
4750042       line[0] = 0;
4750043       //
4750044       // Reset terminal mode.
4750045       //
4750046       terminal_restore (&attr);
4750047       return;
4750048     }
4750049   //
4750050   // Find the last position and, if there is a new
4750051   // line code,
4750052   // replace it with zero. If the string is empty, a
4750053   // ^D was
4750054   // received.
4750055   //
4750056   i = strlen (line);
4750057   if (i > 0 && line[i - 1] == '\n')
4750058     {
4750059       line[i - 1] = '\0';
4750060     }
4750061   //
4750062   // Restore terminal mode.
4750063   //
4750064   terminal_restore (&attr);
4750065 }
4750066
4750067 //-------------------------------------------------------
4750068 static int
4750069 terminal_echo (struct termios *orig)
4750070 {
4750071   int status;
4750072   struct termios attr;
4750073   //
4750074   // Save previous.
4750075   //
4750076   status = tcgetattr (STDIN_FILENO, orig);
4750077   if (status < 0)
4750078     {
4750079       return (-1);
4750080     }
4750081   //
4750082   // Get again.
4750083   //
4750084   status = tcgetattr (STDIN_FILENO, &attr);
4750085   if (status < 0)
4750086     {
4750087       return (-1);
4750088     }
4750089   //
4750090   attr.c_iflag |= (BRKINT | ICRNL);
4750091   attr.c_iflag &= ~(IGNBRK | INLCR);
4750092   //
4750093   attr.c_lflag |=
4750094     (ECHO | ECHOE | ECHOK | ECHONL | ICANON | ISIG);
4750095   attr.c_lflag &= ~(IEXTEN);
4750096   //
4750097   status = tcsetattr (STDIN_FILENO, TCSANOW, &attr);
4750098   //
4750099   return (status);
4750100 }
4750101
4750102 //-------------------------------------------------------
4750103 static int
4750104 terminal_noecho (struct termios *orig)
4750105 {
4750106   int status;
4750107   struct termios attr;
4750108   //
4750109   // Save previous.
4750110   //
4750111   status = tcgetattr (STDIN_FILENO, orig);
4750112   if (status < 0)
4750113     {
4750114       return (-1);
4750115     }
4750116   //
4750117   // Get again.
4750118   //
4750119   status = tcgetattr (STDIN_FILENO, &attr);
4750120   if (status < 0)
4750121     {
4750122       return (-1);
4750123     }
4750124   //
4750125   attr.c_iflag |= (BRKINT | ICRNL);
4750126   attr.c_iflag &= ~(IGNBRK | INLCR);
4750127   //
4750128   attr.c_lflag |= (ICANON | ISIG);
4750129   attr.c_lflag &= ~(ECHO | IEXTEN);
4750130   //
4750131   status = tcsetattr (STDIN_FILENO, TCSANOW, &attr);
4750132   //
4750133   return (status);
4750134 }
4750135
4750136 //-------------------------------------------------------
4750137 static int
4750138 terminal_restore (struct termios *orig)
4750139 {
4750140   int status;
4750141   //
4750142   // For an unknown reason, when running with Bochs,
4750143   // before
4750144   // restoring the termios configuration, the previous
4750145   // one
4750146   // is to be read. Here, 'attr' is just a placeholder
4750147   // and
4750148   // the updated content is not used for anything
4750149   // else.
4750150   //
4750151   struct termios attr;
4750152   status = tcgetattr (STDIN_FILENO, &attr);
4750153   if (status < 0)
4750154     {
4750155       return (-1);
4750156     }
4750157   //
4750158   //
4750159   //
4750160   status = tcsetattr (STDIN_FILENO, TCSANOW, orig);
4750161   //
4750162   return (status);
4750163 }
```

95.21.2 lib/sys/os32/ipconfig.c

Si veda la sezione 87.28.

```
4760001 #include <sys/os32.h>
4760002 #include <errno.h>
4760003 #include <string.h>
4760004 #include <stdio.h>
```

```
4760005 //-----------------------------------------------------------
4760006 int
4760007 ipconfig (int n, in_addr_t address, int m)
4760008 {
4760009   sysmsg_ipconfig_t msg;
4760010   //
4760011   // Fill the message.
4760012   //
4760013   msg.n = n;
4760014   msg.address = address;
4760015   msg.m = m;
4760016   msg.ret = 0;
4760017   //
4760018   // Syscall.
4760019   //
4760020   sys (SYS_IPCONFIG, &msg, (sizeof msg));
4760021   //
4760022   // Check return value.
4760023   //
4760024   if (msg.ret < 0)
4760025     {
4760026       //
4760027       // Something wrong.
4760028       //
4760029       errno = msg.errno;
4760030       errln = msg.errln;
4760031       strncpy (errfn, msg.errfn, PATH_MAX);
4760032     }
4760033   //
4760034   // Return.
4760035   //
4760036   return (msg.ret);
4760037 }
```

95.21.3 lib/sys/os32/mount.c

«

Si veda la sezione 87.36.

```
4770001 #include <sys/types.h>
4770002 #include <errno.h>
4770003 #include <sys/os32.h>
4770004 #include <stddef.h>
4770005 #include <string.h>
4770006 //-----------------------------------------------------------
4770007 int
4770008 mount (const char *path_dev, const char *path_mnt,
4770009        int options)
4770010 {
4770011   sysmsg_mount_t msg;
4770012   //
4770013   msg.path_dev = path_dev;
4770014   msg.path_mnt = path_mnt;
4770015   msg.options = options;
4770016   msg.ret = 0;
4770017   msg.errno = 0;
4770018   //
4770019   sys (SYS_MOUNT, &msg, (sizeof msg));
4770020   //
4770021   errno = msg.errno;
4770022   errln = msg.errln;
4770023   strncpy (errfn, msg.errfn, PATH_MAX);
4770024   return (msg.ret);
4770025 }
```

95.21.4 lib/sys/os32/namep.c

«

Si veda la sezione 88.85.

```
4780001 #include <sys/os32.h>
4780002 #include <stdlib.h>
4780003 #include <errno.h>
4780004 #include <unistd.h>
4780005 //-----------------------------------------------------------
4780006 int
4780007 namep (const char *name, char *path, size_t size)
4780008 {
4780009   char command[PATH_MAX];
4780010   char *env_path;
4780011   int p;        // Index used inside the path
4780012   // environment.
4780013   int c;        // Index used inside the command
4780014   // string.
4780015   int status;
4780016   //
4780017   // Check for valid input.
4780018   //
4780019   if (name == NULL || name[0] == 0 || path == NULL
```



```
4780020       || name == path)
4780021     {
4780022       errset (EINVAL);  // Invalid argument.
4780023       return (-1);
4780024     }
4780025   //
4780026   // Check if the original command contains at least a
4780027   // '/'. Otherwise
4780028   // a scan for the environment variable 'PATH' must
4780029   // be done.
4780030   //
4780031   if (strchr (name, '/') == NULL)
4780032     {
4780033       //
4780034       // Ok: no '/' there. Get the environment
4780035       // variable 'PATH'.
4780036       //
4780037       env_path = getenv ("PATH");
4780038       if (env_path == NULL)
4780039         {
4780040           //
4780041           // There is no 'PATH' environment value.
4780042           //
4780043           errset (ENOENT);      // No such file or
4780044           // directory.
4780045           return (-1);
4780046         }
4780047       //
4780048       // Scan paths and try to find a file with that
4780049       // name.
4780050       //
4780051       for (p = 0; env_path[p] != 0;)
4780052         {
4780053           for (c = 0;
4780054                c < (PATH_MAX - strlen (name) - 2) &&
4780055                env_path[p] != 0 &&
4780056                env_path[p] != ':'; c++, p++)
4780057             {
4780058               command[c] = env_path[p];
4780059             }
4780060           //
4780061           // If the loop is ended because the command
4780062           // array does not
4780063           // have enough room for the full path, then
4780064           // must return an
4780065           // error.
4780066           //
4780067           if (env_path[p] != ':' && env_path[p] != 0)
4780068             {
4780069               errset (ENAMETOOLONG);    // Filename
4780070               // too long.
4780071               return (-1);
4780072             }
4780073           //
4780074           // The command array has enough space. At
4780075           // index 'c' must
4780076           // place a zero, to terminate current
4780077           // string.
4780078           //
4780079           command[c] = 0;
4780080           //
4780081           // Add the rest of the path.
4780082           //
4780083           strcat (command, "/");
4780084           strcat (command, name);
4780085           //
4780086           // Verify to have something with that full
4780087           // path name.
4780088           //
4780089           status = access (command, F_OK);
4780090           if (status == 0)
4780091             {
4780092               //
4780093               // Verify to have enough room inside the
4780094               // destination
4780095               // path.
4780096               //
4780097               if (strlen (command) >= size)
4780098                 {
4780099                   //
4780100                   // Sorry: too big. There must be
4780101                   // room also for
4780102                   // the string termination null
4780103                   // character.
4780104                   //
4780105                   errset (ENAMETOOLONG);        // Filename
4780106                   // too long.
```

```
4780107                     return (-1);
4780108                   }
4780109                 //
4780110                 // Copy the path and return.
4780111                 //
4780112                 strncpy (path, command, size);
4780113                 return (0);
4780114               }
4780115             //
4780116             // That path was not good: try again. But
4780117             // before returning
4780118             // to the external loop, must verify if 'p'
4780119             // is to be
4780120             // incremented, after a ':', because the
4780121             // external loop
4780122             // does not touch the index 'p',
4780123             //
4780124             if (env_path[p] == ':')
4780125               {
4780126                 p++;
4780127               }
4780128           }
4780129         //
4780130         // At this point, there is no match with the
4780131         // paths.
4780132         //
4780133         errset (ENOENT);  // No such file or directory.
4780134         return (-1);
4780135       }
4780136     //
4780137     // At this point, a path was given and the
4780138     // environment variable
4780139     // 'PATH' was not scanned. Just copy the same path.
4780140     // But must verify
4780141     // that the receiving path has enough room for it.
4780142     //
4780143     if (strlen (name) >= size)
4780144       {
4780145         //
4780146         // Sorry: too big.
4780147         //
4780148         errset (ENAMETOOLONG);    // Filename too long.
4780149         return (-1);
4780150       }
4780151     //
4780152     // Ok: copy and return.
4780153     //
4780154     strncpy (path, name, size);
4780155     return (0);
4780156 }
```

95.21.5 lib/sys/os32/routeadd.c

« Si veda la sezione 87.42.

```
4790001 #include <sys/os32.h>
4790002 #include <errno.h>
4790003 #include <string.h>
4790004 #include <stdio.h>
4790005 //-----------------------------------------------------------
4790006 int
4790007 routeadd (in_addr_t destination, int m,
4790008           in_addr_t router, int device)
4790009 {
4790010   sysmsg_route_t msg;
4790011   //
4790012   // Fill the message.
4790013   //
4790014   msg.destination = destination;
4790015   msg.m = m;
4790016   msg.router = router;
4790017   msg.device = device;
4790018   //
4790019   // Syscall.
4790020   //
4790021   sys (SYS_ROUTEADD, &msg, (sizeof msg));
4790022   //
4790023   // Check return value.
4790024   //
4790025   if (msg.ret < 0)
4790026     {
4790027       //
4790028       // Something wrong.
4790029       //
4790030       errno = msg.errno;
4790031       errln = msg.errln;
4790032       strncpy (errfn, msg.errfn, PATH_MAX);
```



```
4790033     }
4790034   //
4790035   // Return.
4790036   //
4790037   return (msg.ret);
4790038 }
```

95.21.6 lib/sys/os32/routedel.c

« Si veda la sezione 87.43.

```
4800001 #include <sys/os32.h>
4800002 #include <errno.h>
4800003 #include <string.h>
4800004 #include <stdio.h>
4800005 //-----------------------------------------------------------
4800006 int
4800007 routedel (in_addr_t destination, int m)
4800008 {
4800009   sysmsg_route_t msg;
4800010   //
4800011   // Fill the message.
4800012   //
4800013   msg.destination = destination;
4800014   msg.m = m;
4800015   //
4800016   // Syscall.
4800017   //
4800018   sys (SYS_ROUTEDEL, &msg, (sizeof msg));
4800019   //
4800020   // Check return value.
4800021   //
4800022   if (msg.ret < 0)
4800023     {
4800024       //
4800025       // Something wrong.
4800026       //
4800027       errno = msg.errno;
4800028       errln = msg.errln;
4800029       strncpy (errfn, msg.errfn, PATH_MAX);
4800030     }
4800031   //
4800032   // Return.
4800033   //
4800034   return (msg.ret);
4800035 }
```

95.21.7 lib/sys/os32/sys.s

« Si veda la sezione 87.56.

```
4810001 .global sys
4810002 #-----------------------------------------------------
4810003 .text
4810004 #-----------------------------------------------------
4810005 # Call a system call.
4810006 #
4810007 # Please remember that system calls should never be
4810008 # used (called) inside the kernel code, because system
4810009 # calls cannot be nested for the os32 simple
4810010 # architecture!
4810011 # If a particular function is necessary inside the
4810012 # kernel, that usually is made by a system call, an
4810013 # appropriate k_...() function must be made, to avoid
4810014 # the problem.
4810015 #
4810016 #-----------------------------------------------------
4810017 .align 4
4810018 sys:
4810019     int   $128  # 0x80
4810020     ret
```

95.21.8 lib/sys/os32/umount.c

« Si veda la sezione 87.36.

```
4820001 #include <sys/types.h>
4820002 #include <errno.h>
4820003 #include <sys/os32.h>
4820004 #include <stddef.h>
4820005 #include <string.h>
4820006 //-----------------------------------------------------------
4820007 int
4820008 umount (const char *path_mnt)
4820009 {
4820010   sysmsg_umount_t msg;
4820011   //
```

```
4820012   msg.path_mnt = path_mnt;
4820013   msg.ret = 0;
4820014   msg.errno = 0;
4820015   //
4820016   sys (SYS_UMOUNT, &msg, (sizeof msg));
4820017   //
4820018   errno = msg.errno;
4820019   errln = msg.errln;
4820020   strncpy (errfn, msg.errfn, PATH_MAX);
4820021   return (msg.ret);
4820022 }
```

95.21.9 lib/sys/os32/z_perror.c

«

Si veda la sezione 87.65.

```
4830001 #include <sys/os32.h>
4830002 #include <errno.h>
4830003 #include <stddef.h>
4830004 #include <string.h>
4830005 //-----------------------------------------------------------
4830006 void
4830007 z_perror (const char *string)
4830008 {
4830009   //
4830010   // If errno is zero, there is nothing to show.
4830011   //
4830012   if (errno == 0)
4830013     {
4830014       return;
4830015     }
4830016   //
4830017   // Show the string if there is one.
4830018   //
4830019   if (string != NULL && strlen (string) > 0)
4830020     {
4830021       z_printf ("%s: ", string);
4830022     }
4830023   //
4830024   // Show the translated error.
4830025   //
4830026   if (errfn[0] != 0 && errln != 0)
4830027     {
4830028       z_printf ("[%s:%u:%i] %s\n",
4830029                 errfn, errln, errno, strerror (errno));
4830030     }
4830031   else
4830032     {
4830033       z_printf ("[%i] %s\n", errno, strerror (errno));
4830034     }
4830035 }
```

95.21.10 lib/sys/os32/z_printf.c

«

Si veda la sezione 87.65.

```
4840001 #include <sys/os32.h>
4840002 #include <restrict.h>
4840003 //-----------------------------------------------------------
4840004 int
4840005 z_printf (const char *restrict format, ...)
4840006 {
4840007   va_list ap;
4840008   va_start (ap, format);
4840009   return z_vprintf (format, ap);
4840010 }
```

95.21.11 lib/sys/os32/z_vprintf.c

«

Si veda la sezione 87.65.

```
4850001 #include <sys/os32.h>
4850002 #include <restrict.h>
4850003 //-----------------------------------------------------------
4850004 int
4850005 z_vprintf (const char *restrict format, va_list arg)
4850006 {
4850007   int ret;
4850008   sysmsg_zpstring_t msg;
4850009   msg.string[0] = 0;
4850010   ret = vsprintf (msg.string, format, arg);
4850011   sys (SYS_ZPSTRING, &msg, (sizeof msg));
4850012   return ret;
4850013 }
```


95.22 os32: «lib/sys/sa_family_t.h»

«

Si veda la sezione 91.3.

```
4860001 #ifndef _SYS_SA_FAMILY_T_H
4860002 #define _SYS_SA_FAMILY_T_H    1
4860003 //-----------------------------------------------------------
4860004 #include <inttypes.h>
4860005 //-----------------------------------------------------------
4860006 typedef uint16_t sa_family_t;   // Address family.
4860007 //-----------------------------------------------------------
4860008 #endif
```

95.23 os32: «lib/sys/socket.h»

«

Si veda la sezione 91.3.

```
4870001 #ifndef _SYS_SOCKET_H
4870002 #define _SYS_SOCKET_H    1
4870003 //-----------------------------------------------------------
4870004 #include <stdint.h>
4870005 #include <unistd.h>
4870006 #include <sys/socklen_t.h>
4870007 #include <sys/sa_family_t.h>
4870008 //-----------------------------------------------------------
4870009 struct sockaddr
4870010 {
4870011   sa_family_t sa_family;        // Address family.
4870012   char sa_data[14];     // Socket address.
4870013 };
4870014 //
4870015 //
4870016 //
4870017 struct sockaddr_storage
4870018 {
4870019   sa_family_t ss_family;        // Socket storage
4870020   // family.
4870021   uint8_t ss_zero[14];  // Filler.
4870022 };
4870023 //
4870024 //
4870025 //
4870026 #define SOCK_STREAM    1        // Byte-stream socket.
4870027 #define SOCK_DGRAM     2        // Datagram socket.
4870028 #define SOCK_RAW       3        // Raw protocol
4870029                                 // interface.
4870030 #define SOCK_SEQPACKET 5        // Sequenced-packet
4870031                                 // socket.
4870032 //
4870033 // Protocol families:
4870034 //
4870035 #define PF_UNSPEC       0       // Unspecified.
4870036 #define PF_UNIX         1       // Unix domain socket.
4870037 #define PF_INET         2       // IPv4 protocol
4870038                                 // family.
4870039 #define PF_INET6        10      // IPv6 protocol
4870040                                 // family.
4870041 //
4870042 // Address families.
4870043 //
4870044 #define AF_UNSPEC  PF_UNSPEC    // Unspecified.
4870045 #define AF_UNIX    PF_UNIX      // Unix domain socket.
4870046 #define AF_INET    PF_INET      // IPv4 address
4870047                                 // family.
4870048 #define AF_INET6   PF_INET6     // IPv6 address
4870049                                 // family.
4870050 //-----------------------------------------------------------
4870051 int accept (int sfdn, struct sockaddr *addr,
4870052             socklen_t * addrlen);
4870053 int bind (int sfdn, const struct sockaddr *addr,
4870054           socklen_t addrlen);
4870055 int connect (int sfdn, const struct sockaddr *addr,
4870056              socklen_t addrlen);
4870057 int listen (int sfdn, int backlog);
4870058 ssize_t send (int sfdn, const void *buffer,
4870059               size_t count, int flags);
4870060 ssize_t recvfrom (int sfdn, void *buffer, size_t count,
4870061                   int flags, struct sockaddr *addrfrom,
4870062                   socklen_t * addrlen);
4870063 int socket (int family, int type, int protocol);
4870064
4870065 #define recv(sdfn, buffer, count, flags) \
4870066     recvfrom (sdfn, buffer, count, flags, NULL, NULL)
4870067 //-----------------------------------------------------------
4870068 #endif
```

95.23.1 lib/sys/socket/accept.c 894
95.23.2 lib/sys/socket/bind.c 895
95.23.3 lib/sys/socket/connect.c 895
95.23.4 lib/sys/socket/listen.c 896
95.23.5 lib/sys/socket/recvfrom.c 896
95.23.6 lib/sys/socket/send.c 898
95.23.7 lib/sys/socket/socket.c 899

95.23.1 lib/sys/socket/accept.c

Si veda la sezione 87.3.

```
4880001 #include <sys/os32.h>
4880002 #include <errno.h>
4880003 #include <string.h>
4880004 #include <stdio.h>
4880005 #include <fcntl.h>
4880006 //-------------------------------------------------------
4880007 int
4880008 accept (int sfdn, struct sockaddr *addr,
4880009         socklen_t * addrlen)
4880010 {
4880011   sysmsg_accept_t msg;
4880012   //
4880013   // Fill the message.
4880014   //
4880015   msg.sfdn = sfdn;
4880016   memset (&msg.addr, 0x00, sizeof (msg.addr));
4880017   msg.addrlen = *addrlen;
4880018   msg.fl_flags = 0;     // Not necessary.
4880019   msg.ret = 0;
4880020   //
4880021   // Syscall.
4880022   //
4880023   while (1)
4880024     {
4880025       sys (SYS_ACCEPT, &msg, (sizeof msg));
4880026       //
4880027       if (msg.ret < 0
4880028           && (msg.errno == EAGAIN
4880029               || msg.errno == EWOULDBLOCK))
4880030         {
4880031           //
4880032           // No request at the moment.
4880033           //
4880034           if (msg.fl_flags & O_NONBLOCK)
4880035             {
4880036               //
4880037               // Don't block.
4880038               //
4880039               break;
4880040             }
4880041           else
4880042             {
4880043               //
4880044               // Keep trying.
4880045               //
4880046               continue;
4880047             }
4880048         }
4880049       else
4880050         {
4880051           break;
4880052         }
4880053     }
4880054   //
4880055   // Check return value.
4880056   //
4880057   if (msg.ret < 0)
4880058     {
4880059       //
4880060       // Something wrong.
4880061       //
4880062       errno = msg.errno;
4880063       errln = msg.errln;
4880064       strncpy (errfn, msg.errfn, PATH_MAX);
4880065     }
4880066   else
4880067     {
4880068       //
4880069       // Update the socket address and the address
4880070       // length.
4880071       //
```



```
4880072       if (addrlen != NULL && addr != NULL && *addrlen > 0)
4880073         {
4880074           memcpy (addr, &msg.addr,
4880075                   min (msg.addrlen, *addrlen));
4880076           *addrlen = msg.addrlen;
4880077         }
4880078     }
4880079   //
4880080   // Return.
4880081   //
4880082   return (msg.ret);
4880083 }
```

95.23.2 lib/sys/socket/bind.c

Si veda la sezione 87.4.

```
4890001 #include <sys/os32.h>
4890002 #include <errno.h>
4890003 #include <string.h>
4890004 #include <stdio.h>
4890005 //-------------------------------------------------------
4890006 int
4890007 bind (int sfdn, const struct sockaddr *addr,
4890008       socklen_t addrlen)
4890009 {
4890010   sysmsg_bind_t msg;
4890011   //
4890012   // Fill the message.
4890013   //
4890014   msg.sfdn = sfdn;
4890015   memcpy (&msg.addr, addr, (size_t) addrlen);
4890016   msg.addrlen = addrlen;
4890017   msg.ret = 0;
4890018   //
4890019   // Syscall.
4890020   //
4890021   sys (SYS_BIND, &msg, (sizeof msg));
4890022   //
4890023   // Check return value.
4890024   //
4890025   if (msg.ret < 0)
4890026     {
4890027       //
4890028       // Something wrong.
4890029       //
4890030       errno = msg.errno;
4890031       errln = msg.errln;
4890032       strncpy (errfn, msg.errfn, PATH_MAX);
4890033     }
4890034   //
4890035   // Return.
4890036   //
4890037   return (msg.ret);
4890038 }
```

95.23.3 lib/sys/socket/connect.c

Si veda la sezione 87.11.

```
4900001 #include <sys/os32.h>
4900002 #include <errno.h>
4900003 #include <string.h>
4900004 #include <stdio.h>
4900005 //-------------------------------------------------------
4900006 int
4900007 connect (int sfdn, const struct sockaddr *addr,
4900008          socklen_t addrlen)
4900009 {
4900010   sysmsg_connect_t msg;
4900011   //
4900012   // Fill the message.
4900013   //
4900014   msg.sfdn = sfdn;
4900015   memcpy (&msg.addr, addr, (size_t) addrlen);
4900016   msg.addrlen = addrlen;
4900017   msg.ret = 0;
4900018   //
4900019   // Syscall.
4900020   //
4900021   while (1)
4900022     {
4900023       sys (SYS_CONNECT, &msg, (sizeof msg));
4900024       //
4900025       if (msg.ret < 0)
4900026         {
4900027           if (msg.errno == EINPROGRESS
```

```
4900028                 || msg.errno == EALREADY)
4900029               {
4900030                 //
4900031                 // Loop until the connection is
4900032                 // established, or a
4900033                 // different error comes.
4900034                 //
4900035                 continue;
4900036               }
4900037             else
4900038               {
4900039                 break;
4900040               }
4900041           }
4900042         else
4900043           {
4900044             break;
4900045           }
4900046       }
4900047     //
4900048     // Check return value.
4900049     //
4900050     if (msg.ret < 0)
4900051       {
4900052         //
4900053         // Something wrong.
4900054         //
4900055         errno = msg.errno;
4900056         errln = msg.errln;
4900057         strncpy (errfn, msg.errfn, PATH_MAX);
4900058       }
4900059     //
4900060     // Return.
4900061     //
4900062     return (msg.ret);
4900063 }
```

95.23.4 lib/sys/socket/listen.c

«

Si veda la sezione 87.31.

```
4910001 #include <sys/os32.h>
4910002 #include <errno.h>
4910003 #include <string.h>
4910004 #include <stdio.h>
4910005 //-------------------------------------------------------
4910006 int
4910007 listen (int sfdn, int backlog)
4910008 {
4910009   sysmsg_listen_t msg;
4910010   //
4910011   // Fill the message.
4910012   //
4910013   msg.sfdn = sfdn;
4910014   msg.backlog = backlog;
4910015   msg.ret = 0;
4910016   //
4910017   // Syscall.
4910018   //
4910019   sys (SYS_LISTEN, &msg, (sizeof msg));
4910020   //
4910021   // Check return value.
4910022   //
4910023   if (msg.ret < 0)
4910024     {
4910025       //
4910026       // Something wrong.
4910027       //
4910028       errno = msg.errno;
4910029       errln = msg.errln;
4910030       strncpy (errfn, msg.errfn, PATH_MAX);
4910031     }
4910032   //
4910033   // Return.
4910034   //
4910035   return (msg.ret);
4910036 }
```

95.23.5 lib/sys/socket/recvfrom.c

«

Si veda la sezione 87.40.

```
4920001 #include <sys/os32.h>
4920002 #include <errno.h>
4920003 #include <string.h>
4920004 #include <stdio.h>
4920005 #include <fcntl.h>
```



```
4920006 //-------------------------------------------------------
4920007 ssize_t
4920008 recvfrom (int sfdn, void *buffer, size_t count,
4920009           int flags, struct sockaddr *addrfrom,
4920010           socklen_t * addrlen)
4920011 {
4920012   sysmsg_recvfrom_t msg;
4920013   //
4920014   // Reduce size of read if necessary.
4920015   //
4920016   if (count > BUFSIZ)
4920017     {
4920018       count = BUFSIZ;
4920019     }
4920020   //
4920021   // Fill the message.
4920022   //
4920023   msg.sfdn = sfdn;
4920024   msg.buffer = buffer;
4920025   msg.count = count;
4920026   msg.flags = flags;
4920027   msg.addrfrom = addrfrom;
4920028   msg.addrsize = addrlen;
4920029   msg.fl_flags = 0;     // Not necessary.
4920030   msg.ret = 0;
4920031   //
4920032   // Repeat syscall, until something is received or
4920033   // end of file is
4920034   // reached.
4920035   //
4920036   while (1)
4920037     {
4920038       sys (SYS_RECVFROM, &msg, (sizeof msg));
4920039       if (msg.ret == 0)
4920040         {
4920041           //
4920042           // Stream closed from the other side.
4920043           //
4920044           break;
4920045         }
4920046       if (msg.ret < 0
4920047           && (msg.errno == EAGAIN
4920048               || msg.errno == EWOULDBLOCK))
4920049         {
4920050           //
4920051           // No data at the moment.
4920052           //
4920053           if (msg.fl_flags & O_NONBLOCK)
4920054             {
4920055               //
4920056               // Don't block.
4920057               //
4920058               break;
4920059             }
4920060           else
4920061             {
4920062               //
4920063               // Keep trying.
4920064               //
4920065               continue;
4920066             }
4920067         }
4920068       //
4920069       // Otherwise, we have received something.
4920070       //
4920071       break;
4920072     }
4920073   //
4920074   //
4920075   //
4920076   if (msg.ret < 0)
4920077     {
4920078       //
4920079       // No valid read.
4920080       //
4920081       errno = msg.errno;
4920082       errln = msg.errln;
4920083       strncpy (errfn, msg.errfn, PATH_MAX);
4920084       return (msg.ret);
4920085     }
4920086   //
4920087   if (msg.ret > count)
4920088     {
4920089       //
4920090       // A strange value was returned. Considering it
4920091       // a read error.
4920092       //
```

```
4920093         errset (EIO);     // I/O error.
4920094         return (-1);
4920095       }
4920096     //
4920097     // A valid read: return.
4920098     //
4920099     return (msg.ret);
4920100   }
```

95.23.6 lib/sys/socket/send.c

«

Si veda la sezione 87.45.

```
4930001   #include <unistd.h>
4930002   #include <sys/os32.h>
4930003   #include <errno.h>
4930004   #include <string.h>
4930005   #include <stdio.h>
4930006   //------------------------------------------------------------
4930007   ssize_t
4930008   send (int sfdn, const void *buffer, size_t count, int flags)
4930009   {
4930010     sysmsg_send_t msg;
4930011     int retry = 3;
4930012     //
4930013     // Reduce size of write if necessary.
4930014     //
4930015     if (count > BUFSIZ)
4930016       {
4930017         count = BUFSIZ;
4930018       }
4930019     //
4930020     // Fill the message.
4930021     //
4930022     msg.sfdn = sfdn;
4930023     msg.buffer = buffer;
4930024     msg.count = count;
4930025     msg.flags = flags;
4930026     //
4930027     // Syscall.
4930028     //
4930029     for (; retry > 0; retry--)
4930030       {
4930031         sys (SYS_SEND, &msg, (sizeof msg));
4930032         //
4930033         // Check.
4930034         //
4930035         if ((msg.ret < 0) && (msg.errno == E_ARP_MISSING))
4930036           {
4930037             sleep (1);
4930038             continue;     // Retry.
4930039           }
4930040         else
4930041           {
4930042             break;
4930043           }
4930044       }
4930045     //
4930046     // Check the final result and return.
4930047     //
4930048     if (msg.ret < 0)
4930049       {
4930050         //
4930051         // No valid write.
4930052         //
4930053         errno = msg.errno;
4930054         errln = msg.errln;
4930055         strncpy (errfn, msg.errfn, PATH_MAX);
4930056         return (msg.ret);
4930057       }
4930058     //
4930059     if (msg.ret > count)
4930060       {
4930061         //
4930062         // A strange value was returned. Considering it
4930063         // a read error.
4930064         //
4930065         errset (EIO);     // I/O error.
4930066         return (-1);
4930067       }
4930068     //
4930069     // A valid write return.
4930070     //
4930071     return (msg.ret);
4930072   }
```


95.23.7 lib/sys/socket/socket.c

«

Si veda la sezione 87.54.

```
4940001   #include <sys/os32.h>
4940002   #include <errno.h>
4940003   #include <string.h>
4940004   #include <stdio.h>
4940005   //------------------------------------------------------------
4940006   int
4940007   socket (int family, int type, int protocol)
4940008   {
4940009     sysmsg_socket_t msg;
4940010     //
4940011     // Fill the message.
4940012     //
4940013     msg.family = family;
4940014     msg.type = type;
4940015     msg.protocol = protocol;
4940016     msg.ret = 0;
4940017     //
4940018     // Syscall.
4940019     //
4940020     sys (SYS_SOCKET, &msg, (sizeof msg));
4940021     //
4940022     // Check return value.
4940023     //
4940024     if (msg.ret < 0)
4940025       {
4940026         //
4940027         // Something wrong.
4940028         //
4940029         errno = msg.errno;
4940030         errln = msg.errln;
4940031         strncpy (errfn, msg.errfn, PATH_MAX);
4940032       }
4940033     //
4940034     // Return.
4940035     //
4940036     return (msg.ret);
4940037   }
```

95.24 os32: «lib/sys/socklen_t.h»

«

Si veda la sezione 91.3.

```
4950001   #ifndef _SYS_SOCKLEN_T_H
4950002   #define _SYS_SOCKLEN_T_H    1
4950003   //------------------------------------------------------------
4950004   #include <stdint.h>
4950005   //------------------------------------------------------------
4950006   typedef uint32_t socklen_t;
4950007   //------------------------------------------------------------
4950008   #endif
```

95.25 os32: «lib/sys/stat.h»

«

Si veda la sezione 91.3.

```
4960001   #ifndef _SYS_STAT_H
4960002   #define _SYS_STAT_H     1
4960003
4960004   #include <restrict.h>
4960005   #include <sys/types.h>  // dev_t
4960006                           // off_t
4960007                           // blkcnt_t
4960008                           // blksize_t
4960009                           // ino_t
4960010                           // mode_t
4960011                           // nlink_t
4960012                           // uid_t
4960013                           // gid_t
4960014                           // time_t
4960015   //------------------------------------------------------------
4960016   // File type.
4960017   //------------------------------------------------------------
4960018   #define S_IFMT   0170000        // File type mask.
4960019   //
4960020   #define S_IFBLK  0060000        // Block device file.
4960021   #define S_IFCHR  0020000        // Character device
4960022                                   // file.
4960023   #define S_IFIFO  0010000        // Pipe (FIFO) file.
4960024   #define S_IFREG  0100000        // Regular file.
4960025   #define S_IFDIR  0040000        // Directory.
4960026   #define S_IFLNK  0120000        // Symbolic link.
4960027   #define S_IFSOCK 0140000        // Unix domain socket.
```

```
4960028 //-------------------------------------------------------
4960029 // Owner user access permissions.
4960030 //-------------------------------------------------------
4960031 #define S_IRWXU  0000700        // Owner user access
4960032                                 // permissions mask.
4960033 //
4960034 #define S_IRUSR  0000400        // Owner user read
4960035                                 // access permission.
4960036 #define S_IWUSR  0000200        // Owner user write
4960037                                 // access permission.
4960038 #define S_IXUSR  0000100        // Owner user
4960039                                 // execution or cross
4960040                                 // perm.
4960041 //-------------------------------------------------------
4960042 // Group owner access permissions.
4960043 //-------------------------------------------------------
4960044 #define S_IRWXG  0000070        // Owner group access
4960045                                 // permissions mask.
4960046 //
4960047 #define S_IRGRP  0000040        // Owner group read
4960048                                 // access permission.
4960049 #define S_IWGRP  0000020        // Owner group write
4960050                                 // access permission.
4960051 #define S_IXGRP  0000010        // Owner group
4960052                                 // execution or cross
4960053                                 // perm.
4960054 //-------------------------------------------------------
4960055 // Other users access permissions.
4960056 //-------------------------------------------------------
4960057 #define S_IRWXO  0000007        // Other users access
4960058                                 // permissions mask.
4960059 //
4960060 #define S_IROTH  0000004        // Other users read
4960061                                 // access permission.
4960062 #define S_IWOTH  0000002        // Other users write
4960063                                 // access permissions.
4960064 #define S_IXOTH  0000001        // Other users
4960065                                 // execution or cross
4960066                                 // perm.
4960067 //-------------------------------------------------------
4960068 // S-bit: in this case there is no mask to select all
4960069 // of them.
4960070 //-------------------------------------------------------
4960071 #define S_ISUID  0004000        // S-UID.
4960072 #define S_ISGID  0002000        // S-GID.
4960073 #define S_ISVTX  0001000        // Sticky.
4960074 //-------------------------------------------------------
4960075 // Macroinstructions to verify the type of file.
4960076 //-------------------------------------------------------
4960077 //
4960078 // Block device:
4960079 //
4960080 #define S_ISBLK(m)   (((m) & S_IFMT) == S_IFBLK)
4960081 //
4960082 // Character device:
4960083 //
4960084 #define S_ISCHR(m)   (((m) & S_IFMT) == S_IFCHR)
4960085 //
4960086 // FIFO.
4960087 //
4960088 #define S_ISFIFO(m)  (((m) & S_IFMT) == S_IFIFO)
4960089 //
4960090 // Regular file.
4960091 //
4960092 #define S_ISREG(m)   (((m) & S_IFMT) == S_IFREG)
4960093 //
4960094 // Directory.
4960095 //
4960096 #define S_ISDIR(m)   (((m) & S_IFMT) == S_IFDIR)
4960097 //
4960098 // Symbolic link.
4960099 //
4960100 #define S_ISLNK(m)   (((m) & S_IFMT) == S_IFLNK)
4960101 //
4960102 // Socket (Unix domain socket).
4960103 //
4960104 #define S_ISSOCK(m)  (((m) & S_IFMT) == S_IFSOCK)
4960105 //-------------------------------------------------------
4960106 // Structure 'stat'.
4960107 //-------------------------------------------------------
4960108 struct stat
4960109 {
4960110   dev_t st_dev; // Device containing the file.
4960111   ino_t st_ino; // File serial number (inode number).
4960112   mode_t st_mode;       // File type and permissions.
4960113   nlink_t st_nlink;     // Links to the file.
4960114   uid_t st_uid; // Owner user id.
```



```
4960115   gid_t st_gid; // Owner group id.
4960116   dev_t st_rdev;        // Device number if it is a
4960117   // device file.
4960118   off_t st_size;        // File size.
4960119   time_t st_atime;      // Last access time.
4960120   time_t st_mtime;      // Last modification time.
4960121   time_t st_ctime;      // Last inode modification.
4960122   blksize_t st_blksize; // Block size for I/O
4960123   // operations.
4960124   blkcnt_t st_blocks;   // File size / block size.
4960125 };
4960126 //-------------------------------------------------------
4960127 // Function prototypes.
4960128 //-------------------------------------------------------
4960129 int chmod (const char *path, mode_t mode);
4960130 int fchmod (int fdn, mode_t mode);
4960131 int fstat (int fdn, struct stat *buffer);
4960132 int lstat (const char *restrict path,
4960133            struct stat *restrict buffer);
4960134 int mkdir (const char *path, mode_t mode);
4960135 int mkfifo (const char *path, mode_t mode);
4960136 int mknod (const char *path, mode_t mode, dev_t dev);
4960137 int stat (const char *restrict path,
4960138           struct stat *restrict buffer);
4960139 mode_t umask (mode_t mask);
4960140
4960141 #endif // _SYS_STAT_H
```


95.25.1 lib/sys/stat/chmod.c 901
95.25.2 lib/sys/stat/fchmod.c 901
95.25.3 lib/sys/stat/fstat.c 902
95.25.4 lib/sys/stat/mkdir.c 902
95.25.5 lib/sys/stat/mknod.c 903
95.25.6 lib/sys/stat/stat.c 903
95.25.7 lib/sys/stat/umask.c 903

95.25.1 lib/sys/stat/chmod.c

Si veda la sezione 87.7.

```
4970001 #include <sys/stat.h>
4970002 #include <string.h>
4970003 #include <sys/os32.h>
4970004 #include <errno.h>
4970005 #include <limits.h>
4970006 //-------------------------------------------------------
4970007 int
4970008 chmod (const char *path, mode_t mode)
4970009 {
4970010   sysmsg_chmod_t msg;
4970011   //
4970012   msg.path = path;
4970013   msg.mode = mode;
4970014   //
4970015   sys (SYS_CHMOD, &msg, (sizeof msg));
4970016   //
4970017   errno = msg.errno;
4970018   errln = msg.errln;
4970019   strncpy (errfn, msg.errfn, PATH_MAX);
4970020   return (msg.ret);
4970021 }
```

95.25.2 lib/sys/stat/fchmod.c

Si veda la sezione 87.7.

```
4980001 #include <sys/stat.h>
4980002 #include <string.h>
4980003 #include <sys/os32.h>
4980004 #include <errno.h>
4980005 #include <limits.h>
4980006 //-------------------------------------------------------
4980007 int
4980008 fchmod (int fdn, mode_t mode)
4980009 {
4980010   sysmsg_fchmod_t msg;
4980011   //
4980012   msg.fdn = fdn;
4980013   msg.mode = mode;
4980014   //
4980015   sys (SYS_FCHMOD, &msg, (sizeof msg));
4980016   //
```

```
4980017   errno = msg.errno;
4980018   errrln = msg.errln;
4980019   strncpy (errfn, msg.errfn, PATH_MAX);
4980020   return (msg.ret);
4980021 }
```

95.25.3 lib/sys/stat/fstat.c

Si veda la sezione 87.55.

```
4990001 #include <unistd.h>
4990002 #include <errno.h>
4990003 #include <sys/os32.h>
4990004 #include <string.h>
4990005 //-------------------------------------------------------
4990006 int
4990007 fstat (int fdn, struct stat *buffer)
4990008 {
4990009   sysmsg_fstat_t msg;
4990010   //
4990011   msg.fdn = fdn;
4990012   msg.stat.st_dev = buffer->st_dev;
4990013   msg.stat.st_ino = buffer->st_ino;
4990014   msg.stat.st_mode = buffer->st_mode;
4990015   msg.stat.st_nlink = buffer->st_nlink;
4990016   msg.stat.st_uid = buffer->st_uid;
4990017   msg.stat.st_gid = buffer->st_gid;
4990018   msg.stat.st_rdev = buffer->st_rdev;
4990019   msg.stat.st_size = buffer->st_size;
4990020   msg.stat.st_atime = buffer->st_atime;
4990021   msg.stat.st_mtime = buffer->st_mtime;
4990022   msg.stat.st_ctime = buffer->st_ctime;
4990023   msg.stat.st_blksize = buffer->st_blksize;
4990024   msg.stat.st_blocks = buffer->st_blocks;
4990025   //
4990026   sys (SYS_FSTAT, &msg, (sizeof msg));
4990027   //
4990028   buffer->st_dev = msg.stat.st_dev;
4990029   buffer->st_ino = msg.stat.st_ino;
4990030   buffer->st_mode = msg.stat.st_mode;
4990031   buffer->st_nlink = msg.stat.st_nlink;
4990032   buffer->st_uid = msg.stat.st_uid;
4990033   buffer->st_gid = msg.stat.st_gid;
4990034   buffer->st_rdev = msg.stat.st_rdev;
4990035   buffer->st_size = msg.stat.st_size;
4990036   buffer->st_atime = msg.stat.st_atime;
4990037   buffer->st_mtime = msg.stat.st_mtime;
4990038   buffer->st_ctime = msg.stat.st_ctime;
4990039   buffer->st_blksize = msg.stat.st_blksize;
4990040   buffer->st_blocks = msg.stat.st_blocks;
4990041   //
4990042   errno = msg.errno;
4990043   errrln = msg.errln;
4990044   strncpy (errfn, msg.errfn, PATH_MAX);
4990045   return (msg.ret);
4990046 }
```

95.25.4 lib/sys/stat/mkdir.c

Si veda la sezione 87.34.

```
5000001 #include <sys/stat.h>
5000002 #include <string.h>
5000003 #include <sys/os32.h>
5000004 #include <errno.h>
5000005 #include <limits.h>
5000006 //-------------------------------------------------------
5000007 int
5000008 mkdir (const char *path, mode_t mode)
5000009 {
5000010   sysmsg_mkdir_t msg;
5000011   //
5000012   msg.path = path;
5000013   msg.mode = mode;
5000014   //
5000015   sys (SYS_MKDIR, &msg, (sizeof msg));
5000016   //
5000017   errno = msg.errno;
5000018   errrln = msg.errln;
5000019   strncpy (errfn, msg.errfn, PATH_MAX);
5000020   return (msg.ret);
5000021 }
```


95.25.5 lib/sys/stat/mknod.c

Si veda la sezione 87.35.

```
5010001 #include <unistd.h>
5010002 #include <errno.h>
5010003 #include <sys/os32.h>
5010004 #include <string.h>
5010005 //-------------------------------------------------------
5010006 int
5010007 mknod (const char *path, mode_t mode, dev_t device)
5010008 {
5010009   sysmsg_mknod_t msg;
5010010   //
5010011   msg.path = path;
5010012   msg.mode = mode;
5010013   msg.device = device;
5010014   //
5010015   sys (SYS_MKNOD, &msg, (sizeof msg));
5010016   //
5010017   errno = msg.errno;
5010018   errrln = msg.errln;
5010019   strncpy (errfn, msg.errfn, PATH_MAX);
5010020   return (msg.ret);
5010021 }
```

95.25.6 lib/sys/stat/stat.c

Si veda la sezione 87.55.

```
5020001 #include <unistd.h>
5020002 #include <errno.h>
5020003 #include <sys/os32.h>
5020004 #include <string.h>
5020005 //-------------------------------------------------------
5020006 int
5020007 stat (const char *path, struct stat *buffer)
5020008 {
5020009   sysmsg_stat_t msg;
5020010   //
5020011   msg.path = path;
5020012   //
5020013   msg.stat.st_dev = buffer->st_dev;
5020014   msg.stat.st_ino = buffer->st_ino;
5020015   msg.stat.st_mode = buffer->st_mode;
5020016   msg.stat.st_nlink = buffer->st_nlink;
5020017   msg.stat.st_uid = buffer->st_uid;
5020018   msg.stat.st_gid = buffer->st_gid;
5020019   msg.stat.st_rdev = buffer->st_rdev;
5020020   msg.stat.st_size = buffer->st_size;
5020021   msg.stat.st_atime = buffer->st_atime;
5020022   msg.stat.st_mtime = buffer->st_mtime;
5020023   msg.stat.st_ctime = buffer->st_ctime;
5020024   msg.stat.st_blksize = buffer->st_blksize;
5020025   msg.stat.st_blocks = buffer->st_blocks;
5020026   //
5020027   sys (SYS_STAT, &msg, (sizeof msg));
5020028   //
5020029   buffer->st_dev = msg.stat.st_dev;
5020030   buffer->st_ino = msg.stat.st_ino;
5020031   buffer->st_mode = msg.stat.st_mode;
5020032   buffer->st_nlink = msg.stat.st_nlink;
5020033   buffer->st_uid = msg.stat.st_uid;
5020034   buffer->st_gid = msg.stat.st_gid;
5020035   buffer->st_rdev = msg.stat.st_rdev;
5020036   buffer->st_size = msg.stat.st_size;
5020037   buffer->st_atime = msg.stat.st_atime;
5020038   buffer->st_mtime = msg.stat.st_mtime;
5020039   buffer->st_ctime = msg.stat.st_ctime;
5020040   buffer->st_blksize = msg.stat.st_blksize;
5020041   buffer->st_blocks = msg.stat.st_blocks;
5020042   //
5020043   errno = msg.errno;
5020044   errrln = msg.errln;
5020045   strncpy (errfn, msg.errfn, PATH_MAX);
5020046   return (msg.ret);
5020047 }
```

95.25.7 lib/sys/stat/umask.c

Si veda la sezione 87.60.

```
5030001 #include <sys/stat.h>
5030002 #include <string.h>
5030003 #include <sys/os32.h>
5030004 #include <errno.h>
5030005 #include <limits.h>
5030006 //-------------------------------------------------------
```

```
5030007 mode_t
5030008 umask (mode_t mask)
5030009 {
5030010   sysmsg_umask_t msg;
5030011   msg.umask = mask;
5030012   sys (SYS_UMASK, &msg, (sizeof msg));
5030013   return (msg.ret);
5030014 }
```

95.26 os32: «lib/sys/types.h»

Si veda la sezione 91.3.

```
5040001 #ifndef _SYS_TYPES_H
5040002 #define _SYS_TYPES_H    1
5040003 //-------------------------------------------------------------
5040004 #include <clock_t.h>
5040005 #include <time_t.h>
5040006 #include <size_t.h>
5040007 //-------------------------------------------------------------
5040008 typedef long int blkcnt_t;
5040009 typedef long int blksize_t;
5040010 typedef uint16_t dev_t; // Traditional device size.
5040011 typedef unsigned int id_t;
5040012 typedef unsigned int gid_t;
5040013 typedef unsigned int uid_t;
5040014 typedef uint16_t ino_t; // Minix 1 file system inode
5040015                         // size.
5040016 typedef uint16_t mode_t;        // Minix 1 file system
5040017                                 // mode size.
5040018 typedef unsigned int nlink_t;
5040019 typedef long long int off_t;
5040020 typedef int pid_t;
5040021 typedef unsigned long int pthread_t;
5040022 typedef int ssize_t;
5040023 //-------------------------------------------------------------
5040024 // Common extentions.
5040025 //
5040026 dev_t makedev (int major, int minor);
5040027 int major (dev_t device);
5040028 int minor (dev_t device);
5040029 //-------------------------------------------------------------
5040030 #endif
```

95.26.1 lib/sys/types/major.c 904
95.26.2 lib/sys/types/makedev.c 904
95.26.3 lib/sys/types/minor.c 904

95.26.1 lib/sys/types/major.c

Si veda la sezione 88.75.

```
5050001 #include <sys/types.h>
5050002 //-------------------------------------------------------------
5050003 int
5050004 major (dev_t device)
5050005 {
5050006   return ((int) (device / 256));
5050007 }
```

95.26.2 lib/sys/types/makedev.c

Si veda la sezione 88.75.

```
5060001 #include <sys/types.h>
5060002 //-------------------------------------------------------------
5060003 dev_t
5060004 makedev (int major, int minor)
5060005 {
5060006   return ((dev_t) (major * 256 + minor));
5060007 }
```

95.26.3 lib/sys/types/minor.c

Si veda la sezione 88.75.

```
5070001 #include <sys/types.h>
5070002 //-------------------------------------------------------------
5070003 int
5070004 minor (dev_t device)
5070005 {
5070006   return ((dev_t) (device & 0x00FF));
5070007 }
```


95.27 os32: «lib/sys/wait.h»

Si veda la sezione 91.3.

```
5080001 #ifndef _SYS_WAIT_H
5080002 #define _SYS_WAIT_H      1
5080003
5080004 #include <sys/types.h>
5080005
5080006 //-------------------------------------------------------------
5080007 pid_t wait (int *status);
5080008 //-------------------------------------------------------------
5080009
5080010 #endif
```

95.27.1 lib/sys/wait/wait.c 905

95.27.1 lib/sys/wait/wait.c

Si veda la sezione 87.63.

```
5090001 #include <sys/types.h>
5090002 #include <errno.h>
5090003 #include <sys/os32.h>
5090004 #include <stddef.h>
5090005 #include <string.h>
5090006 //-------------------------------------------------------------
5090007 pid_t
5090008 wait (int *status)
5090009 {
5090010   sysmsg_wait_t msg;
5090011   msg.ret = 0;
5090012   msg.errno = 0;
5090013   msg.status = 0;
5090014   while (msg.ret == 0)
5090015     {
5090016       //
5090017       // Loop as long as there are children, an none
5090018       // is dead.
5090019       //
5090020       sys (SYS_WAIT, &msg, (sizeof msg));
5090021     }
5090022   errno = msg.errno;
5090023   errln = msg.errln;
5090024   strncpy (errfn, msg.errfn, PATH_MAX);
5090025   //
5090026   if (status != NULL)
5090027     {
5090028       //
5090029       // Only the low eight bits are returned.
5090030       //
5090031       *status = (msg.status & 0x00FF);
5090032     }
5090033   return (msg.ret);
5090034 }
```

95.28 os32: «lib/termios.h»

Si veda la sezione 87.58.

```
5100001 #ifndef _TERMIOS_H
5100002 #define _TERMIOS_H       1
5100003 //-------------------------------------------------------------
5100004 #include <stdint.h>
5100005 //-------------------------------------------------------------
5100006 typedef uint16_t tcflag_t;
5100007 typedef unsigned char cc_t;
5100008 //-------------------------------------------------------------
5100009 #define NCCS    11      // 'c_cc[]' size.
5100010 //
5100011 struct termios
5100012 {
5100013   tcflag_t c_iflag;
5100014   tcflag_t c_oflag;
5100015   tcflag_t c_cflag;
5100016   tcflag_t c_lflag;
5100017   cc_t c_cc[NCCS];
5100018 };
5100019 //
5100020 // Subscript names for 'c_cc[]' array, inside the
5100021 // 'termios' structure:
5100022 //
5100023 #define VEOF      0     // EOF character
5100024 #define VEOL      1     // EOL character
5100025 #define VERASE    2     // ERASE character
5100026 #define VINTR     3     // INTR character
5100027 #define VKILL     4     // KILL character
5100028 #define VMIN      5     // MIN value
```

```
#define VQUIT     6     // QUIT character
#define VSTART    7     // START character
#define VSTOP     8     // STOP character
#define VSUSP     9     // SUSP character
#define VTIME    10     // TIME value
//
// Input modes, for 'c_iflag' inside the 'termios'
// structure:
//
#define BRKINT    1     // signal interrupt on break
#define ICRNL     2     // map CR to NL on input
#define IGNBRK    4     // ignore break condition
#define IGNCR     8     // ignore CR
#define IGNPAR   16     // ignore characters with
                        // parity errors
#define INLCR    32     // map NL to CR on input
#define INPCK    64     // enable input parity check
#define ISTRIP  128     // strip off eighth bit
#define IXOFF   256     // enable start/stop input
                        // control
#define IXON    512     // enable start/stop output
                        // control
#define PARMRK 1024     // mark parity errors
//
// Output modes, for 'c_oflag' inside the 'termios'
// structure:
//
#define OPOST     1     // post-process output
//
// Control modes, for 'c_cflag' inside the 'termios'
// structure:
//   not implemented.
//
//
// Local modes, for 'c_lflag' inside the 'termios'
// structure:
//
#define ECHO      1     // enable echo
#define ECHOE     2     // echo erase character as
                        // backspace
#define ECHOK     4     // echo KILL
#define ECHONL    8     // echo NL
#define ICANON   16     // canonical input mode
#define IEXTEN   32     // extended input mode
#define ISIG     64     // enable signals
#define NOFLSH  128     // disable flush after
                        // interrupt or quit
#define TOSTOP  256     // send SIGTTOU for background
                        // output
//-----------------------------------------------------
// Optional action for use with 'tcsetattr()':
//
#define TCSANOW   1     // change attributes
                        // immediately
#define TCSADRAIN 2     // change attributes when
                        // output has drained
#define TCSAFLUSH 3     // change attributes when
                        // output has drained,
                        // and also flush pending
                        // input
//-----------------------------------------------------
int tcgetattr (int fdn, struct termios *termios_p);
int tcsetattr (int fdn, int action,
               struct termios *termios_p);
//-----------------------------------------------------
#endif
```

95.28.1 lib/termios/tcgetattr.c 906
95.28.2 lib/termios/tcsetattr.c 907

95.28.1 lib/termios/tcgetattr.c

«

Si veda la sezione 87.58.

```
#include <termios.h>
#include <sys/os32.h>
#include <errno.h>
//-----------------------------------------------------
#define DEBUG 0
//-----------------------------------------------------
int
tcgetattr (int fdn, struct termios *termios_p)
{
  sysmsg_tcattr_t msg;
  msg.fdn = fdn;
  msg.attr = termios_p;
```



```
  sys (SYS_TCGETATTR, &msg, (sizeof msg));
  errno = msg.errno;
  errln = msg.errln;
  strncpy (errfn, msg.errfn, PATH_MAX);
  return (msg.ret);
}
```

95.28.2 lib/termios/tcsetattr.c

«

Si veda la sezione 87.58.

```
#include <termios.h>
#include <sys/os32.h>
#include <errno.h>
//-----------------------------------------------------
#define DEBUG 0
//-----------------------------------------------------
int
tcsetattr (int fdn, int action, struct termios *termios_p)
{
  sysmsg_tcattr_t msg;
  msg.fdn = fdn;
  msg.action = action;
  msg.attr = termios_p;
  sys (SYS_TCSETATTR, &msg, (sizeof msg));
  errno = msg.errno;
  errln = msg.errln;
  strncpy (errfn, msg.errfn, PATH_MAX);
  return (msg.ret);
}
```

95.29 os32: «lib/time.h»

«

Si veda la sezione 91.3.

```
#ifndef _TIME_H
#define _TIME_H       1
//-----------------------------------------------------
#include <restrict.h>
#include <size_t.h>
#include <time_t.h>
#include <clock_t.h>
#include <NULL.h>
#include <stdint.h>
//-----------------------------------------------------
#define CLOCKS_PER_SEC  ((clock_t) 100)
//-----------------------------------------------------
struct tm
{
  int tm_sec;
  int tm_min;
  int tm_hour;
  int tm_mday;
  int tm_mon;
  int tm_year;
  int tm_wday;
  int tm_yday;
  int tm_isdst;
};
//-----------------------------------------------------
clock_t clock (void);
time_t time (time_t * timer);
int stime (time_t * timer);
double difftime (time_t time1, time_t time0);
time_t mktime (const struct tm *timeptr);
struct tm *gmtime (const time_t * timer);
struct tm *localtime (const time_t * timer);
char *asctime (const struct tm *timeptr);
char *ctime (const time_t * timer);
size_t strftime (char *restrict s, size_t maxsize,
                 const char *restrict format,
                 const struct tm *restrict timeptr);
//-----------------------------------------------------
#define difftime(t1,t0) ((double)((t1)-(t0)))
#define ctime(t)        (asctime (localtime (t)))
#define localtime(t)    (gmtime (t))
//-----------------------------------------------------
#endif
```

95.29.1 lib/time/asctime.c 908
95.29.2 lib/time/clock.c 909
95.29.3 lib/time/gmtime.c 909
95.29.4 lib/time/mktime.c 911

95.29.5 lib/time/stime.c 913
95.29.6 lib/time/time.c 913

95.29.1 lib/time/asctime.c

Si veda la sezione 88.15.

```
5140001 #include <time.h>
5140002 #include <string.h>
5140003 #include <stdio.h>
5140004 //-----------------------------------------------------------
5140005 char *
5140006 asctime (const struct tm *timeptr)
5140007 {
5140008   static char time_string[25];  // 'Sun Jan 30
5140009   // 24:00:00 2111'
5140010   //
5140011   // Check argument.
5140012   //
5140013   if (timeptr == NULL)
5140014     {
5140015       return (NULL);
5140016     }
5140017   //
5140018   // Set week day.
5140019   //
5140020   switch (timeptr->tm_wday)
5140021     {
5140022     case 0:
5140023       strcpy (&time_string[0], "Sun");
5140024       break;
5140025     case 1:
5140026       strcpy (&time_string[0], "Mon");
5140027       break;
5140028     case 2:
5140029       strcpy (&time_string[0], "Tue");
5140030       break;
5140031     case 3:
5140032       strcpy (&time_string[0], "Wed");
5140033       break;
5140034     case 4:
5140035       strcpy (&time_string[0], "Thu");
5140036       break;
5140037     case 5:
5140038       strcpy (&time_string[0], "Fri");
5140039       break;
5140040     case 6:
5140041       strcpy (&time_string[0], "Sat");
5140042       break;
5140043     default:
5140044       strcpy (&time_string[0], "Err");
5140045     }
5140046   //
5140047   // Set month.
5140048   //
5140049   switch (timeptr->tm_mon)
5140050     {
5140051     case 1:
5140052       strcpy (&time_string[3], " Jan");
5140053       break;
5140054     case 2:
5140055       strcpy (&time_string[3], " Feb");
5140056       break;
5140057     case 3:
5140058       strcpy (&time_string[3], " Mar");
5140059       break;
5140060     case 4:
5140061       strcpy (&time_string[3], " Apr");
5140062       break;
5140063     case 5:
5140064       strcpy (&time_string[3], " May");
5140065       break;
5140066     case 6:
5140067       strcpy (&time_string[3], " Jun");
5140068       break;
5140069     case 7:
5140070       strcpy (&time_string[3], " Jul");
5140071       break;
5140072     case 8:
5140073       strcpy (&time_string[3], " Aug");
5140074       break;
5140075     case 9:
5140076       strcpy (&time_string[3], " Sep");
5140077       break;
5140078     case 10:
5140079       strcpy (&time_string[3], " Oct");
5140080       break;
5140081     case 11:
5140082       strcpy (&time_string[3], " Nov");
5140083       break;
5140084     case 12:
5140085       strcpy (&time_string[3], " Dec");
5140086       break;
5140087     default:
5140088       strcpy (&time_string[3], " Err");
5140089     }
5140090   //
5140091   // Set day of month, hour, minute, second and year.
5140092   //
5140093   sprintf (&time_string[7], " %2i %2i:%2i:%2i %4i",
5140094            timeptr->tm_mday, timeptr->tm_hour,
5140095            timeptr->tm_min, timeptr->tm_sec,
5140096            timeptr->tm_year);
5140097   //
5140098   //
5140099   //
5140100   return (&time_string[0]);
5140101 }
```


95.29.2 lib/time/clock.c

Si veda la sezione 87.9.

```
5150001 #include <time.h>
5150002 #include <sys/os32.h>
5150003 //-----------------------------------------------------------
5150004 clock_t
5150005 clock (void)
5150006 {
5150007   sysmsg_clock_t msg;
5150008   msg.ret = 0;
5150009   sys (SYS_CLOCK, &msg, (sizeof msg));
5150010   return (msg.ret);
5150011 }
```

95.29.3 lib/time/gmtime.c

Si veda la sezione 88.15.

```
5160001 #include <time.h>
5160002 //-----------------------------------------------------------
5160003 static int leap_year (int year);
5160004 //-----------------------------------------------------------
5160005 struct tm *
5160006 gmtime (const time_t * timer)
5160007 {
5160008   static struct tm tms;
5160009   int loop;
5160010   unsigned int remainder;
5160011   unsigned int days;
5160012   //
5160013   // Check argument.
5160014   //
5160015   if (timer == NULL)
5160016     {
5160017       return (NULL);
5160018     }
5160019   //
5160020   // Days since epoch. There are 86400 seconds per
5160021   // day.
5160022   // At the moment, the field 'tm_yday' will contain
5160023   // all days since epoch.
5160024   //
5160025   days = *timer / 86400L;
5160026   remainder = *timer % 86400L;
5160027   //
5160028   // Minutes, after full days.
5160029   //
5160030   tms.tm_min = remainder / 60U;
5160031   //
5160032   // Seconds, after full minutes.
5160033   //
5160034   tms.tm_sec = remainder % 60U;
5160035   //
5160036   // Hours, after full days.
5160037   //
5160038   tms.tm_hour = tms.tm_min / 60;
5160039   //
5160040   // Minutes, after full hours.
5160041   //
5160042   tms.tm_min = tms.tm_min % 60;
5160043   //
5160044   // Find the week day. Must remove some days to align
5160045   // the
```

```
5160046   // calculation. So: the week days of the first week
5160047   // of 1970
5160048   // are not valid! After 1970-01-04 calculations are
5160049   // right.
5160050   //
5160051   tms.tm_wday = (days - 3) % 7;
5160052   //
5160053   // Find the year: the field 'tm_yday' will be
5160054   // reduced to the days
5160055   // of current year.
5160056   //
5160057   for (tms.tm_year = 1970; days > 0; tms.tm_year++)
5160058     {
5160059       if (leap_year (tms.tm_year))
5160060         {
5160061           if (days >= 366)
5160062             {
5160063               days -= 366;
5160064               continue;
5160065             }
5160066           else
5160067             {
5160068               break;
5160069             }
5160070         }
5160071       else
5160072         {
5160073           if (days >= 365)
5160074             {
5160075               days -= 365;
5160076               continue;
5160077             }
5160078           else
5160079             {
5160080               break;
5160081             }
5160082         }
5160083     }
5160084   //
5160085   // Day of the year.
5160086   //
5160087   tms.tm_yday = days + 1;
5160088   //
5160089   // Find the month.
5160090   //
5160091   tms.tm_mday = days + 1;
5160092   //
5160093   for (tms.tm_mon = 0, loop = 1; tms.tm_mon <= 12 && loop;)
5160094     {
5160095       tms.tm_mon++;
5160096       //
5160097       switch (tms.tm_mon)
5160098         {
5160099         case 1:
5160100         case 3:
5160101         case 5:
5160102         case 7:
5160103         case 8:
5160104         case 10:
5160105         case 12:
5160106           if (tms.tm_mday >= 31)
5160107             {
5160108               tms.tm_mday -= 31;
5160109             }
5160110           else
5160111             {
5160112               loop = 0;
5160113             }
5160114           break;
5160115         case 4:
5160116         case 6:
5160117         case 9:
5160118         case 11:
5160119           if (tms.tm_mday >= 30)
5160120             {
5160121               tms.tm_mday -= 30;
5160122             }
5160123           else
5160124             {
5160125               loop = 0;
5160126             }
5160127           break;
5160128         case 2:
5160129           if (leap_year (tms.tm_year))
5160130             {
5160131               if (tms.tm_mday >= 29)
5160132                 {
5160133                   tms.tm_mday -= 29;
5160134                 }
5160135               else
5160136                 {
5160137                   loop = 0;
5160138                 }
5160139             }
5160140           else
5160141             {
5160142               if (tms.tm_mday >= 28)
5160143                 {
5160144                   tms.tm_mday -= 28;
5160145                 }
5160146               else
5160147                 {
5160148                   loop = 0;
5160149                 }
5160150             }
5160151           break;
5160152         }
5160153     }
5160154   //
5160155   // No check for day light saving time.
5160156   //
5160157   tms.tm_isdst = 0;
5160158   //
5160159   // Return.
5160160   //
5160161   return (&tms);
5160162 }
5160163
5160164 //-----------------------------------------------------------
5160165 static int
5160166 leap_year (int year)
5160167 {
5160168   if ((year % 4) == 0)
5160169     {
5160170       if ((year % 100) == 0)
5160171         {
5160172           if ((year % 400) == 0)
5160173             {
5160174               return (1);
5160175             }
5160176           else
5160177             {
5160178               return (0);
5160179             }
5160180         }
5160181       else
5160182         {
5160183           return (1);
5160184         }
5160185     }
5160186   else
5160187     {
5160188       return (0);
5160189     }
5160190 }
```


95.29.4 lib/time/mktime.c

«

Si veda la sezione 88.15.

```
5170001 #include <time.h>
5170002 #include <string.h>
5170003 #include <stdio.h>
5170004 //-----------------------------------------------------------
5170005 static int leap_year (int year);
5170006 //-----------------------------------------------------------
5170007 time_t
5170008 mktime (const struct tm *timeptr)
5170009 {
5170010   time_t timer_total;
5170011   time_t timer_aux;
5170012   int days;
5170013   int month;
5170014   int year;
5170015   //
5170016   // From seconds to days.
5170017   //
5170018   timer_total = timeptr->tm_sec;
5170019   //
5170020   timer_aux = timeptr->tm_min;
5170021   timer_aux *= 60;
5170022   timer_total += timer_aux;
5170023   //
5170024   timer_aux = timeptr->tm_hour;
```

```
5170025   timer_aux *= (60 * 60);
5170026   timer_total += timer_aux;
5170027   //
5170028   timer_aux = timeptr->tm_mday;
5170029   timer_aux *= 24;
5170030   timer_aux *= (60 * 60);
5170031   timer_total += timer_aux;
5170032   //
5170033   // Month: add the days of months.
5170034   // Will scan the months, from the first, but before
5170035   // the
5170036   // months of the value inside field 'tm_mon'.
5170037   //
5170038   for (month = 1, days = 0; month < timeptr->tm_mon;
5170039        month++)
5170040     {
5170041       switch (month)
5170042         {
5170043         case 1:
5170044         case 3:
5170045         case 5:
5170046         case 7:
5170047         case 8:
5170048         case 10:
5170049           //
5170050           // There is no December, because the scan
5170051           // can go up to
5170052           // the month before the value inside field
5170053           // 'tm_mon'.
5170054           //
5170055           days += 31;
5170056           break;
5170057         case 4:
5170058         case 6:
5170059         case 9:
5170060         case 11:
5170061           days += 30;
5170062           break;
5170063         case 2:
5170064           if (leap_year (timeptr->tm_year))
5170065             {
5170066               days += 29;
5170067             }
5170068           else
5170069             {
5170070               days += 28;
5170071             }
5170072           break;
5170073         }
5170074     }
5170075   //
5170076   timer_aux = days;
5170077   timer_aux *= 24;
5170078   timer_aux *= (60 * 60);
5170079   timer_total += timer_aux;
5170080   //
5170081   // Year. The work is similar to the one of months:
5170082   // days of
5170083   // years are counted, up to the year before the one
5170084   // reported
5170085   // by the field 'tm_year'.
5170086   //
5170087   for (year = 1970, days = 0; year < timeptr->tm_year;
5170088        year++)
5170089     {
5170090       if (leap_year (year))
5170091         {
5170092           days += 366;
5170093         }
5170094       else
5170095         {
5170096           days += 365;
5170097         }
5170098     }
5170099   //
5170100   // After all, must subtract a day from the total.
5170101   //
5170102   days--;
5170103   //
5170104   timer_aux = days;
5170105   timer_aux *= 24;
5170106   timer_aux *= (60 * 60);
5170107   timer_total += timer_aux;
5170108   //
5170109   // That's all.
5170110   //
5170111   return (timer_total);
```



```
5170112 }
5170113
5170114 //-----------------------------------------------------------
5170115 int
5170116 leap_year (int year)
5170117 {
5170118   if ((year % 4) == 0)
5170119     {
5170120       if ((year % 100) == 0)
5170121         {
5170122           if ((year % 400) == 0)
5170123             {
5170124               return (1);
5170125             }
5170126           else
5170127             {
5170128               return (0);
5170129             }
5170130         }
5170131       else
5170132         {
5170133           return (1);
5170134         }
5170135     }
5170136   else
5170137     {
5170138       return (0);
5170139     }
5170140 }
```

95.29.5 lib/time/stime.c

Si veda la sezione 87.59.

```
5180001 #include <time.h>
5180002 #include <sys/os32.h>
5180003 #include <errno.h>
5180004 //-----------------------------------------------------------
5180005 int
5180006 stime (time_t * timer)
5180007 {
5180008   sysmsg_stime_t msg;
5180009   //
5180010   if (timer == NULL)
5180011     {
5180012       errset (EINVAL);
5180013       return (-1);
5180014     }
5180015   //
5180016   msg.timer = *timer;
5180017   msg.ret = 0;
5180018   sys (SYS_STIME, &msg, (sizeof msg));
5180019   return (msg.ret);
5180020 }
```

95.29.6 lib/time/time.c

Si veda la sezione 87.59.

```
5190001 #include <time.h>
5190002 #include <sys/os32.h>
5190003 //-----------------------------------------------------------
5190004 time_t
5190005 time (time_t * timer)
5190006 {
5190007   sysmsg_time_t msg;
5190008   msg.ret = ((time_t) 0);
5190009   sys (SYS_TIME, &msg, (sizeof msg));
5190010   if (timer != NULL)
5190011     {
5190012       *timer = msg.ret;
5190013     }
5190014   return (msg.ret);
5190015 }
```

95.30 os32: «lib/unistd.h»

Si veda la sezione 91.3.

```
5200001 #ifndef _UNISTD_H
5200002 #define _UNISTD_H       1
5200003 //-----------------------------------------------------------
5200004 #include <sys/stat.h>
5200005 #include <sys/types.h>  // size_t, ssize_t, uid_t,
5200006                         // gid_t, off_t, pid_t
5200007 #include <inttypes.h>   // intptr_t
```

```
#include <SEEK.h>       // SEEK_CUR, SEEK_SET,
                        // SEEK_END
//---------------------------------------------------------
typedef unsigned int useconds_t;        // This type
                                        // should be
                                        // used for
                                        // the
                                  // obsolete function
                                  // 'usleep()', that
                                  // is only
                                  // implemented inside
                                  // the
                                  // kernel, as
                                  // 'k_usleep()', for
                                  // the
                                  // drivers
                                  // management.
//---------------------------------------------------------
extern char **environ;  // Variable 'environ' is used
                        // by functions like
                        // 'execv()' in replacement
                        // for 'envp[][]'.
//---------------------------------------------------------
extern char *optarg;    // Used by 'optarg()'.
extern int optind;      //
extern int opterr;      //
extern int optopt;      //
//---------------------------------------------------------
#define STDIN_FILENO    0       //
#define STDOUT_FILENO   1       // Standard file
                                // descriptors.
#define STDERR_FILENO   2       //
//---------------------------------------------------------
#define R_OK            4       // Read permission.
#define W_OK            2       // Write permission.
#define X_OK            1       // Execute or traverse
                                // permission.
#define F_OK            0       // File exists.
//---------------------------------------------------------

int access (const char *path, int mode);
int brk (void *address);
int chdir (const char *path);
int chown (const char *path, uid_t uid, gid_t gid);
int close (int fdn);
int dup (int fdn_old);
int dup2 (int fdn_old, int fdn_new);
int execl (const char *path, char *arg, ...);
int execle (const char *path, char *arg, ...);
int execlp (const char *path, char *arg, ...);
int execv (const char *path, char *const argv[]);
int execve (const char *path, char *const argv[],
            char *const envp[]);
int execvp (const char *path, char *const argv[]);
void _exit (int status);
int fchown (int fdn, uid_t uid, gid_t gid);
pid_t fork (void);
char *getcwd (char *buffer, size_t size);
gid_t getegid (void);
uid_t geteuid (void);
gid_t getgid (void);
int getopt (int argc, char *const argv[],
            const char *optstring);
pid_t getpgrp (void);
pid_t getppid (void);
pid_t getpid (void);
uid_t getuid (void);
int isatty (int fdn);
int link (const char *path_old, const char *path_new);
off_t lseek (int fdn, off_t offset, int whence);
#define      nice(n)     (0)
int pipe (int pipefd[2]);
ssize_t read (int fdn, void *buffer, size_t count);
#define      readlink(p,b,s) ((ssize_t) -1)
int rmdir (const char *path);
void *sbrk (intptr_t increment);
int setegid (gid_t gid);
int seteuid (uid_t uid);
int setgid (gid_t gid);
int setpgrp (void);
int setuid (uid_t uid);
unsigned int sleep (unsigned int s);
#define      sync() /**/
char *ttyname (int fdn);
int unlink (const char *path);
ssize_t write (int fdn, const void *buffer, size_t count);
//---------------------------------------------------------
```



```
#endif
```


95.30.1 lib/unistd/_exit.c 915
95.30.2 lib/unistd/access.c 916
95.30.3 lib/unistd/brk.c 916
95.30.4 lib/unistd/chdir.c 917
95.30.5 lib/unistd/chown.c 917
95.30.6 lib/unistd/close.c 917
95.30.7 lib/unistd/dup.c 918
95.30.8 lib/unistd/dup2.c 918
95.30.9 lib/unistd/environ.c 918
95.30.10 lib/unistd/execl.c 918
95.30.11 lib/unistd/execle.c 919
95.30.12 lib/unistd/execlp.c 919
95.30.13 lib/unistd/execv.c 920
95.30.14 lib/unistd/execve.c 920
95.30.15 lib/unistd/execvp.c 921
95.30.16 lib/unistd/fchdir.c 922
95.30.17 lib/unistd/fchown.c 922
95.30.18 lib/unistd/fork.c 922
95.30.19 lib/unistd/getcwd.c 923
95.30.20 lib/unistd/getegid.c 923
95.30.21 lib/unistd/geteuid.c 924
95.30.22 lib/unistd/getgid.c 924
95.30.23 lib/unistd/getopt.c 924
95.30.24 lib/unistd/getpgrp.c 927
95.30.25 lib/unistd/getpid.c 927
95.30.26 lib/unistd/getppid.c 928
95.30.27 lib/unistd/getuid.c 928
95.30.28 lib/unistd/isatty.c 928
95.30.29 lib/unistd/link.c 929
95.30.30 lib/unistd/lseek.c 929
95.30.31 lib/unistd/pipe.c 929
95.30.32 lib/unistd/read.c 930
95.30.33 lib/unistd/rmdir.c 931
95.30.34 lib/unistd/sbrk.c 931
95.30.35 lib/unistd/setegid.c 932
95.30.36 lib/unistd/seteuid.c 932
95.30.37 lib/unistd/setgid.c 932
95.30.38 lib/unistd/setpgrp.c 932
95.30.39 lib/unistd/setuid.c 933
95.30.40 lib/unistd/sleep.c 933
95.30.41 lib/unistd/ttyname.c 933
95.30.42 lib/unistd/unlink.c 934
95.30.43 lib/unistd/write.c 934

95.30.1 lib/unistd/_exit.c

Si veda la sezione 87.2.

```
#include <unistd.h>
#include <sys/os32.h>
//---------------------------------------------------------
void
_exit (int status)
{
  sysmsg_exit_t msg;
```

```
5210008   //
5210009   // Only the low eight bit are returned.
5210010   //
5210011   msg.status = (status & 0xFF);
5210012   //
5210013   //
5210014   //
5210015   sys (SYS_EXIT, &msg, (sizeof msg));
5210016   //
5210017   // Should not return from system call, but if it
5210018   // does, loop
5210019   // forever:
5210020   //
5210021   while (1);
5210022 }
```

95.30.2 lib/unistd/access.c

«

Si veda la sezione 88.4.

```
5220001 #include <unistd.h>
5220002 #include <sys/stat.h>
5220003 #include <errno.h>
5220004 //-----------------------------------------------------------
5220005 int
5220006 access (const char *path, int mode)
5220007 {
5220008   struct stat st;
5220009   int status;
5220010   uid_t euid;
5220011   //
5220012   status = stat (path, &st);
5220013   if (status != 0)
5220014     {
5220015       return (-1);
5220016     }
5220017   //
5220018   // File exists?
5220019   //
5220020   if (mode == F_OK)
5220021     {
5220022       return (0);
5220023     }
5220024   //
5220025   // Some access permissions are requested: get
5220026   // effective user id.
5220027   //
5220028   euid = geteuid ();
5220029   //
5220030   // Check owner access permissions.
5220031   //
5220032   if (st.st_uid == euid
5220033       && ((st.st_mode & S_IRWXU) == (mode << 6)))
5220034     {
5220035       return (0);
5220036     }
5220037   //
5220038   // Check others access permissions.
5220039   //
5220040   if ((st.st_mode & S_IRWXO) == (mode))
5220041     {
5220042       return (0);
5220043     }
5220044   //
5220045   // Otherwise there are no access permissions.
5220046   //
5220047   errset (EACCES);      // Permission denied.
5220048   return (-1);
5220049 }
```

95.30.3 lib/unistd/brk.c

«

Si veda la sezione 87.5.

```
5230001 #include <unistd.h>
5230002 #include <string.h>
5230003 #include <sys/os32.h>
5230004 #include <errno.h>
5230005 #include <limits.h>
5230006 //-----------------------------------------------------------
5230007 int
5230008 brk (void *address)
5230009 {
5230010   sysmsg_brk_t msg;
5230011   //
5230012   if (address == NULL)
5230013     {
```



```
5230014       errset (EINVAL);
5230015       return (-1);
5230016     }
5230017   //
5230018   msg.address = address;
5230019   //
5230020   sys (SYS_BRK, &msg, (sizeof msg));
5230021   //
5230022   errno = msg.errno;
5230023   errln = msg.errln;
5230024   strncpy (errfn, msg.errfn, PATH_MAX);
5230025   return (msg.ret);
5230026 }
```

95.30.4 lib/unistd/chdir.c

«

Si veda la sezione 87.6.

```
5240001 #include <unistd.h>
5240002 #include <string.h>
5240003 #include <sys/os32.h>
5240004 #include <errno.h>
5240005 #include <limits.h>
5240006 //-----------------------------------------------------------
5240007 int
5240008 chdir (const char *path)
5240009 {
5240010   sysmsg_chdir_t msg;
5240011   //
5240012   msg.path = path;
5240013   msg.ret = 0;
5240014   msg.errno = 0;
5240015   //
5240016   sys (SYS_CHDIR, &msg, (sizeof msg));
5240017   //
5240018   errno = msg.errno;
5240019   errln = msg.errln;
5240020   strncpy (errfn, msg.errfn, PATH_MAX);
5240021   return (msg.ret);
5240022 }
```

95.30.5 lib/unistd/chown.c

«

Si veda la sezione 87.8.

```
5250001 #include <unistd.h>
5250002 #include <string.h>
5250003 #include <sys/os32.h>
5250004 #include <errno.h>
5250005 #include <limits.h>
5250006 //-----------------------------------------------------------
5250007 int
5250008 chown (const char *path, uid_t uid, gid_t gid)
5250009 {
5250010   sysmsg_chown_t msg;
5250011   //
5250012   msg.path = path;
5250013   msg.uid = uid;
5250014   msg.gid = gid;
5250015   //
5250016   sys (SYS_CHOWN, &msg, (sizeof msg));
5250017   //
5250018   errno = msg.errno;
5250019   errln = msg.errln;
5250020   strncpy (errfn, msg.errfn, PATH_MAX);
5250021   return (msg.ret);
5250022 }
```

95.30.6 lib/unistd/close.c

«

Si veda la sezione 87.10.

```
5260001 #include <unistd.h>
5260002 #include <errno.h>
5260003 #include <sys/os32.h>
5260004 #include <string.h>
5260005 //-----------------------------------------------------------
5260006 int
5260007 close (int fdn)
5260008 {
5260009   sysmsg_close_t msg;
5260010   msg.fdn = fdn;
5260011   //
5260012   while (1)
5260013     {
5260014       sys (SYS_CLOSE, &msg, (sizeof msg));
5260015       if (msg.ret < 0 && (msg.errno == EINPROGRESS
```

```
                                || msg.errno == EALREADY))
          {
            continue;
          }
        //
        break;
      }
    errno = msg.errno;
    errln = msg.errln;
    strncpy (errfn, msg.errfn, PATH_MAX);
    return (msg.ret);
}
```

95.30.7 lib/unistd/dup.c

Si veda la sezione 87.12.

```
#include <unistd.h>
#include <sys/os32.h>
#include <string.h>
#include <errno.h>
//-------------------------------------------------------------
int
dup (int fdn_old)
{
  sysmsg_dup_t msg;
  //
  msg.fdn_old = fdn_old;
  //
  sys (SYS_DUP, &msg, (sizeof msg));
  //
  errno = msg.errno;
  errln = msg.errln;
  strncpy (errfn, msg.errfn, PATH_MAX);
  return (msg.ret);
}
```

95.30.8 lib/unistd/dup2.c

Si veda la sezione 87.12.

```
#include <unistd.h>
#include <sys/os32.h>
#include <string.h>
#include <errno.h>
//-------------------------------------------------------------
int
dup2 (int fdn_old, int fdn_new)
{
  sysmsg_dup2_t msg;
  //
  msg.fdn_old = fdn_old;
  msg.fdn_new = fdn_new;
  //
  sys (SYS_DUP2, &msg, (sizeof msg));
  //
  errno = msg.errno;
  errln = msg.errln;
  strncpy (errfn, msg.errfn, PATH_MAX);
  return (msg.ret);
}
```

95.30.9 lib/unistd/environ.c

Si veda la sezione 91.1.

```
#include <unistd.h>
//-------------------------------------------------------------
char **environ;
```

95.30.10 lib/unistd/execl.c

Si veda la sezione 88.21.

```
#include <unistd.h>
#include <limits.h>
#include <stdarg.h>
#include <stddef.h>
//-------------------------------------------------------------
int
execl (const char *path, char *arg, ...)
{
  int argc;
  char *arg_next;
  char *argv[ARG_MAX / 2];
  //
```



```
  va_list ap;
  va_start (ap, arg);
  //
  arg_next = arg;
  //
  for (argc = 0; argc < ARG_MAX / 2; argc++)
    {
      argv[argc] = arg_next;
      if (argv[argc] == NULL)
        {
          break;        // End of arguments.
        }
      arg_next = va_arg (ap, char *);
    }
  //
  return (execve (path, argv, environ));        // [1]
}

//
// The variable 'environ' is declared as
// 'char **environ' and is
// included from <unistd.h>.
//
```

95.30.11 lib/unistd/execle.c

Si veda la sezione 88.21.

```
#include <unistd.h>
#include <limits.h>
#include <stdarg.h>
#include <stddef.h>
//-------------------------------------------------------------
int
execle (const char *path, char *arg, ...)
{
  int argc;
  char *arg_next;
  char *argv[ARG_MAX / 2];
  char **envp;
  //
  va_list ap;
  va_start (ap, arg);
  //
  arg_next = arg;
  //
  for (argc = 0; argc < ARG_MAX / 2; argc++)
    {
      argv[argc] = arg_next;
      if (argv[argc] == NULL)
        {
          break;        // End of arguments.
        }
      arg_next = va_arg (ap, char *);
    }
  //
  envp = va_arg (ap, char **);
  //
  return (execve (path, argv, envp));
}
```

95.30.12 lib/unistd/execlp.c

Si veda la sezione 88.21.

```
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/os32.h>
//-------------------------------------------------------------
int
execlp (const char *path, char *arg, ...)
{
  int argc;
  char *arg_next;
  char *argv[ARG_MAX / 2];
  char command[PATH_MAX];
  int status;
  //
  va_list ap;
  va_start (ap, arg);
  //
  arg_next = arg;
  //
  for (argc = 0; argc < ARG_MAX / 2; argc++)
    {
```

```
5320023       argv[argc] = arg_next;
5320024       if (argv[argc] == NULL)
5320025         {
5320026           break;        // End of arguments.
5320027         }
5320028       arg_next = va_arg (ap, char *);
5320029     }
5320030   //
5320031   // Get a full command path if necessary.
5320032   //
5320033   status = namep (path, command, (size_t) PATH_MAX);
5320034   if (status != 0)
5320035     {
5320036       //
5320037       // Variable 'errno' is already set by
5320038       // 'commandp()'.
5320039       //
5320040       return (-1);
5320041     }
5320042   //
5320043   // Return calling 'execve()'
5320044   //
5320045   return (execve (command, argv, environ));     // [1]
5320046 }
5320047
5320048 //
5320049 // The variable 'environ' is declared as
5320050 // 'char **environ' and is
5320051 // included from <unistd.h>.
5320052 //
```

95.30.13 lib/unistd/execv.c

«

Si veda la sezione 88.21.

```
5330001 #include <unistd.h>
5330002 //-------------------------------------------------------
5330003 int
5330004 execv (const char *path, char *const argv[])
5330005 {
5330006   return (execve (path, argv, environ));        // [1]
5330007 }
5330008
5330009 //
5330010 // The variable 'environ' is declared as
5330011 // 'char **environ' and is
5330012 // included from <unistd.h>.
5330013 //
```

95.30.14 lib/unistd/execve.c

«

Si veda la sezione 87.14.

```
5340001 #include <unistd.h>
5340002 #include <sys/types.h>
5340003 #include <sys/os32.h>
5340004 #include <errno.h>
5340005 #include <string.h>
5340006 #include <string.h>
5340007 //-------------------------------------------------------
5340008 int
5340009 execve (const char *path, char *const argv[],
5340010         char *const envp[])
5340011 {
5340012   sysmsg_exec_t msg;
5340013   size_t size;
5340014   size_t arg_size;
5340015   int argc;
5340016   size_t env_size;
5340017   int envc;
5340018   char *arg_data = msg.arg_data;
5340019   char *env_data = msg.env_data;
5340020   //
5340021   msg.path = path;
5340022   msg.ret = 0;
5340023   msg.errno = 0;
5340024   //
5340025   // Copy 'argv[]' inside a the message buffer
5340026   // 'msg.arg_data',
5340027   // separating each string with a null character and
5340028   // counting the
5340029   // number of strings inside 'argc'.
5340030   //
5340031   for (argc = 0, arg_size = 0, size = 0;
5340032        argv != NULL &&
5340033        argc < (ARG_MAX / 16) &&
5340034        arg_size < ARG_MAX / 2 &&
5340035        argv[argc] != NULL; argc++, arg_size += size)
5340036     {
5340037       size = strlen (argv[argc]);
5340038       size++;   // Count also the final null
5340039       // character.
5340040       if (size > (ARG_MAX / 2 - arg_size))
5340041         {
5340042           errset (E2BIG);       // Argument list too
5340043           // long.
5340044           return (-1);
5340045         }
5340046       strncpy (arg_data, argv[argc], size);
5340047       arg_data += size;
5340048     }
5340049   msg.argc = argc;
5340050   //
5340051   // Copy 'envp[]' inside a the message buffer
5340052   // 'msg.env_data',
5340053   // separating each string with a null character and
5340054   // counting the
5340055   // number of strings inside 'envc'.
5340056   //
5340057   for (envc = 0, env_size = 0, size = 0;
5340058        envp != NULL &&
5340059        envc < (ARG_MAX / 16) &&
5340060        env_size < ARG_MAX / 2 &&
5340061        envp[envc] != NULL; envc++, env_size += size)
5340062     {
5340063       size = strlen (envp[envc]);
5340064       size++;   // Count also the final null
5340065       // character.
5340066       if (size > (ARG_MAX / 2 - env_size))
5340067         {
5340068           errset (E2BIG);       // Argument list too
5340069           // long.
5340070           return (-1);
5340071         }
5340072       strncpy (env_data, envp[envc], size);
5340073       env_data += size;
5340074     }
5340075   msg.envc = envc;
5340076   //
5340077   // System call.
5340078   //
5340079   sys (SYS_EXEC, &msg, (sizeof msg));
5340080   //
5340081   // Should not return, but if it does, then there is
5340082   // an error.
5340083   //
5340084   errno = msg.errno;
5340085   errln = msg.errln;
5340086   strncpy (errfn, msg.errfn, PATH_MAX);
5340087   return (msg.ret);
5340088 }
```

95.30.15 lib/unistd/execvp.c

«

Si veda la sezione 88.21.

```
5350001 #include <unistd.h>
5350002 #include <string.h>
5350003 #include <stdlib.h>
5350004 #include <errno.h>
5350005 #include <sys/os32.h>
5350006 //-------------------------------------------------------
5350007 int
5350008 execvp (const char *path, char *const argv[])
5350009 {
5350010   char command[PATH_MAX];
5350011   int status;
5350012   //
5350013   // Get a full command path if necessary.
5350014   //
5350015   status = namep (path, command, (size_t) PATH_MAX);
5350016   if (status != 0)
5350017     {
5350018       //
5350019       // Variable 'errno' is already set by 'namep()'.
5350020       //
5350021       return (-1);
5350022     }
5350023   //
5350024   // Return calling 'execve()'
5350025   //
5350026   return (execve (command, argv, environ));     // [1]
5350027 }
5350028
```

```
5350029   //
5350030   // The variable 'environ' is declared as
5350031   // 'char **environ' and is
5350032   // included from <unistd.h>.
5350033   //
```

95.30.16 lib/unistd/fchdir.c

Si veda la sezione 87.6.

```
5360001 #include <unistd.h>
5360002 #include <errno.h>
5360003 //-------------------------------------------------------
5360004 int
5360005 fchdir (int fdn)
5360006 {
5360007   //
5360008   // os32 requires to keep track of the path for the
5360009   // current working
5360010   // directory. The standard function 'fchdir()' is
5360011   // not applicable.
5360012   //
5360013   errset (E_NOT_IMPLEMENTED);
5360014   return (-1);
5360015 }
```

95.30.17 lib/unistd/fchown.c

Si veda la sezione 87.8.

```
5370001 #include <unistd.h>
5370002 #include <string.h>
5370003 #include <sys/os32.h>
5370004 #include <errno.h>
5370005 #include <limits.h>
5370006 //-------------------------------------------------------
5370007 int
5370008 fchown (int fdn, uid_t uid, gid_t gid)
5370009 {
5370010   sysmsg_fchown_t msg;
5370011   //
5370012   msg.fdn = fdn;
5370013   msg.uid = uid;
5370014   msg.gid = gid;
5370015   //
5370016   sys (SYS_FCHOWN, &msg, (sizeof msg));
5370017   //
5370018   errno = msg.errno;
5370019   errln = msg.errln;
5370020   strncpy (errfn, msg.errfn, PATH_MAX);
5370021   return (msg.ret);
5370022 }
```

95.30.18 lib/unistd/fork.c

Si veda la sezione 87.19.

```
5380001 #include <unistd.h>
5380002 #include <sys/types.h>
5380003 #include <sys/os32.h>
5380004 #include <errno.h>
5380005 #include <string.h>
5380006 //-------------------------------------------------------
5380007 pid_t
5380008 fork (void)
5380009 {
5380010   sysmsg_fork_t msg;
5380011   //
5380012   // Set the return value for the child process.
5380013   //
5380014   msg.ret = 0;
5380015   //
5380016   // Do the system call.
5380017   //
5380018   sys (SYS_FORK, &msg, (sizeof msg));
5380019   //
5380020   // If the system call has successfully generated a
5380021   // copy of
5380022   // the original process, the following code is
5380023   // executed from
5380024   // the parent and the child. But the child has the
5380025   // 'msg'
5380026   // structure untouched, while the parent has, at
5380027   // least, the
5380028   // pid number inside 'msg.ret'.
5380029   // If the system call fails, there is no child, and
5380030   // the
```



```
5380031   // parent finds the return value equal to -1, with
5380032   // an
5380033   // error number.
5380034   //
5380035   errno = msg.errno;
5380036   errln = msg.errln;
5380037   strncpy (errfn, msg.errfn, PATH_MAX);
5380038   return (msg.ret);
5380039 }
```

95.30.19 lib/unistd/getcwd.c

Si veda la sezione 87.21.

```
5390001 #include <unistd.h>
5390002 #include <sys/types.h>
5390003 #include <sys/os32.h>
5390004 #include <errno.h>
5390005 #include <stddef.h>
5390006 #include <string.h>
5390007 //-------------------------------------------------------
5390008 char *
5390009 getcwd (char *buffer, size_t size)
5390010 {
5390011   sysmsg_uarea_t msg;
5390012   //
5390013   // Check arguments: the buffer must be given.
5390014   //
5390015   if (buffer == NULL)
5390016     {
5390017       errset (EINVAL);
5390018       return (NULL);
5390019     }
5390020   //
5390021   msg.path_cwd = buffer;
5390022   msg.path_cwd_size = size;
5390023   //
5390024   // Set the last buffer element to zero, for later
5390025   // verification.
5390026   //
5390027   buffer[size - 1] = 0;
5390028   //
5390029   // Just get the user area data.
5390030   //
5390031   sys (SYS_UAREA, &msg, (sizeof msg));
5390032   //
5390033   // Check that the path is still correctly
5390034   // terminated. If it isn't,
5390035   // the path is longer than the buffer size, because
5390036   // the last null
5390037   // character was overwritten.
5390038   //
5390039   if (buffer[size - 1] != 0)
5390040     {
5390041       errset (ERANGE);
5390042       return (NULL);
5390043     }
5390044   //
5390045   // Everything is fine.
5390046   //
5390047   return (buffer);
5390048 }
```

95.30.20 lib/unistd/getegid.c

Si veda la sezione 87.22.

```
5400001 #include <unistd.h>
5400002 #include <sys/types.h>
5400003 #include <sys/os32.h>
5400004 #include <errno.h>
5400005 //-------------------------------------------------------
5400006 gid_t
5400007 getegid (void)
5400008 {
5400009   sysmsg_uarea_t msg;
5400010   msg.path_cwd = NULL;
5400011   msg.path_cwd_size = 0;
5400012   sys (SYS_UAREA, &msg, (sizeof msg));
5400013   return (msg.egid);
5400014 }
```

95.30.21 lib/unistd/geteuid.c

«

Si veda la sezione 87.27.

```
5410001 #include <unistd.h>
5410002 #include <sys/types.h>
5410003 #include <sys/os32.h>
5410004 #include <errno.h>
5410005 //-------------------------------------------------------
5410006 uid_t
5410007 geteuid (void)
5410008 {
5410009   sysmsg_uarea_t msg;
5410010   msg.path_cwd = NULL;
5410011   msg.path_cwd_size = 0;
5410012   sys (SYS_UAREA, &msg, (sizeof msg));
5410013   return (msg.euid);
5410014 }
```

95.30.22 lib/unistd/getgid.c

«

Si veda la sezione 87.22.

```
5420001 #include <unistd.h>
5420002 #include <sys/types.h>
5420003 #include <sys/os32.h>
5420004 #include <errno.h>
5420005 //-------------------------------------------------------
5420006 gid_t
5420007 getgid (void)
5420008 {
5420009   sysmsg_uarea_t msg;
5420010   msg.path_cwd = NULL;
5420011   msg.path_cwd_size = 0;
5420012   sys (SYS_UAREA, &msg, (sizeof msg));
5420013   return (msg.gid);
5420014 }
```

95.30.23 lib/unistd/getopt.c

«

Si veda la sezione 88.56.

```
5430001 #include <unistd.h>
5430002 #include <sys/types.h>
5430003 #include <sys/os32.h>
5430004 #include <errno.h>
5430005 //-------------------------------------------------------
5430006 char *optarg;
5430007 int optind = 1;
5430008 int opterr = 1;
5430009 int optopt = 0;
5430010 //-------------------------------------------------------
5430011 static void getopt_no_argument (int opt);
5430012 //-------------------------------------------------------
5430013 int
5430014 getopt (int argc, char *const argv[], const char *optstring)
5430015 {
5430016   static int o = 0;     // Index to scan grouped
5430017   // options.
5430018   int s;        // Index to scan 'optstring'
5430019   int opt;      // Current option letter.
5430020   int flag_argument;    // If there should be an
5430021   // argument.
5430022   //
5430023   // Entering the function, 'flag_argument' is zero.
5430024   // Just to make
5430025   // it clear:
5430026   //
5430027   flag_argument = 0;
5430028   //
5430029   // Scan 'argv[]' elements, starting form the value
5430030   // that 'optind'
5430031   // already have.
5430032   //
5430033   for (; optind < argc; optind++)
5430034     {
5430035       //
5430036       // If an option is expected, some check must be
5430037       // done at
5430038       // the beginning.
5430039       //
5430040       if (!flag_argument)
5430041         {
5430042           //
5430043           // Check if the scan is finished and
5430044           // 'optind' should be kept
5430045           // untouched:
5430046           // 'argv[optind]' is a null pointer;
5430047           // 'argv[optind][0]' is not the character
5430048           // '-';
5430049           // 'argv[optind]' points to the string "-";
5430050           // all 'argv[]' elements are parsed.
5430051           //
5430052           if (argv[optind] == NULL
5430053               || argv[optind][0] != '-'
5430054               || argv[optind][1] == 0 || optind >= argc)
5430055             {
5430056               return (-1);
5430057             }
5430058           //
5430059           // Check if the scan is finished and
5430060           // 'optind' is to be
5430061           // incremented:
5430062           // 'argv[optind]' points to the string "--".
5430063           //
5430064           if (argv[optind][0] == '-'
5430065               && argv[optind][1] == '-'
5430066               && argv[optind][2] == 0)
5430067             {
5430068               optind++;
5430069               return (-1);
5430070             }
5430071         }
5430072       //
5430073       // Scan 'argv[optind]' using the static index
5430074       // 'o'.
5430075       //
5430076       for (; o < strlen (argv[optind]); o++)
5430077         {
5430078           //
5430079           // If there should be an option, index 'o'
5430080           // should
5430081           // start from 1, because 'argv[optind][0]'
5430082           // must
5430083           // be equal to '-'.
5430084           //
5430085           if (!flag_argument && (o == 0))
5430086             {
5430087               //
5430088               // As there is no options, 'o' cannot
5430089               // start
5430090               // from zero, so a new loop is done.
5430091               //
5430092               continue;
5430093             }
5430094           //
5430095           if (flag_argument)
5430096             {
5430097               //
5430098               // There should be an argument, starting
5430099               // from
5430100               // 'argv[optind][o]'.
5430101               //
5430102               if ((o == 0) && (argv[optind][o] == '-'))
5430103                 {
5430104                   //
5430105                   // 'argv[optind][0]' is equal to
5430106                   // '-', but there
5430107                   // should be an argument instead:
5430108                   // the argument
5430109                   // is missing.
5430110                   //
5430111                   optarg = NULL;
5430112                   //
5430113                   if (optstring[0] == ':')
5430114                     {
5430115                       //
5430116                       // As the option string starts
5430117                       // with ':' the
5430118                       // function must return ':'.
5430119                       //
5430120                       optopt = opt;
5430121                       opt = ':';
5430122                     }
5430123                   else
5430124                     {
5430125                       //
5430126                       // As the option string does not
5430127                       // start with ':'
5430128                       // the function must return '?'.
5430129                       //
5430130                       getopt_no_argument (opt);
5430131                       optopt = opt;
5430132                       opt = '?';
5430133                     }
```

```
5430134                 //
5430135                 // 'optind' is left untouched.
5430136                 //
5430137               }
5430138             else
5430139               {
5430140                 //
5430141                 // The argument is found: 'optind'
5430142                 // is to be
5430143                 // incremented and 'o' is reset.
5430144                 //
5430145                 optarg = &argv[optind][o];
5430146                 optind++;
5430147                 o = 0;
5430148               }
5430149             //
5430150             // Return the option, or ':', or '?'.
5430151             //
5430152             return (opt);
5430153           }
5430154         else
5430155           {
5430156             //
5430157             // It should be an option: 'optstring[]'
5430158             // must be
5430159             // scanned.
5430160             //
5430161             opt = argv[optind][o];
5430162             //
5430163             for (s = 0, optopt = 0;
5430164                  s < strlen (optstring); s++)
5430165               {
5430166                 //
5430167                 // If 'optsting[0]' is equal to ':',
5430168                 // index 's' must
5430169                 // start at 1.
5430170                 //
5430171                 if ((s == 0) && (optstring[0] == ':'))
5430172                   {
5430173                     continue;
5430174                   }
5430175                 //
5430176                 if (opt == optstring[s])
5430177                   {
5430178                     //
5430179                     if (optstring[s + 1] == ':')
5430180                       {
5430181                         //
5430182                         // There is an argument.
5430183                         //
5430184                         flag_argument = 1;
5430185                         break;
5430186                       }
5430187                     else
5430188                       {
5430189                         //
5430190                         // There is no argument.
5430191                         //
5430192                         o++;
5430193                         return (opt);
5430194                       }
5430195                   }
5430196               }
5430197             //
5430198             if (s >= strlen (optstring))
5430199               {
5430200                 //
5430201                 // The 'optstring' scan is concluded
5430202                 // with no
5430203                 // match.
5430204                 //
5430205                 o++;
5430206                 optopt = opt;
5430207                 return ('?');
5430208               }
5430209             //
5430210             // Otherwise the loop was broken.
5430211             //
5430212           }
5430213       }
5430214     //
5430215     // Check index 'o'.
5430216     //
5430217     if (o >= strlen (argv[optind]))
5430218       {
5430219         //
5430220         // There are no more options or there is no
```

```
5430221         // argument
5430222         // inside current 'argv[optind]' string.
5430223         // Index 'o' is
5430224         // reset before the next loop.
5430225         //
5430226         o = 0;
5430227       }
5430228   }
5430229 //
5430230 // No more elements inside 'argv' or loop broken:
5430231 // there might be a
5430232 // missing argument.
5430233 //
5430234 if (flag_argument)
5430235   {
5430236     //
5430237     // Missing option argument.
5430238     //
5430239     optarg = NULL;
5430240     //
5430241     if (optstring[0] == ':')
5430242       {
5430243         return (':');
5430244       }
5430245     else
5430246       {
5430247         getopt_no_argument (opt);
5430248         return ('?');
5430249       }
5430250   }
5430251 //
5430252 return (-1);
5430253 }
5430254
5430255 //-------------------------------------------------------------
5430256 static void
5430257 getopt_no_argument (int opt)
5430258 {
5430259   if (opterr)
5430260     {
5430261       fprintf (stderr,
5430262                "Missing argument for option '-%c'\n", opt);
5430263     }
5430264 }
```

95.30.24 lib/unistd/getpgrp.c

Si veda la sezione 87.25.

```
5440001 #include <unistd.h>
5440002 #include <sys/types.h>
5440003 #include <sys/os32.h>
5440004 #include <errno.h>
5440005 //-------------------------------------------------------------
5440006 pid_t
5440007 getpgrp (void)
5440008 {
5440009   sysmsg_uarea_t msg;
5440010   msg.path_cwd = NULL;
5440011   msg.path_cwd_size = 0;
5440012   sys (SYS_UAREA, &msg, (sizeof msg));
5440013   return (msg.pgrp);
5440014 }
```

95.30.25 lib/unistd/getpid.c

Si veda la sezione 87.25.

```
5450001 #include <unistd.h>
5450002 #include <sys/types.h>
5450003 #include <sys/os32.h>
5450004 #include <errno.h>
5450005 //-------------------------------------------------------------
5450006 pid_t
5450007 getpid (void)
5450008 {
5450009   sysmsg_uarea_t msg;
5450010   msg.path_cwd = NULL;
5450011   msg.path_cwd_size = 0;
5450012   sys (SYS_UAREA, &msg, (sizeof msg));
5450013   return (msg.pid);
5450014 }
```

95.30.26 lib/unistd/getppid.c

Si veda la sezione 87.25.

```
5460001 #include <unistd.h>
5460002 #include <sys/types.h>
5460003 #include <sys/os32.h>
5460004 #include <errno.h>
5460005 //-----------------------------------------------------------
5460006 pid_t
5460007 getppid (void)
5460008 {
5460009   sysmsg_uarea_t msg;
5460010   msg.path_cwd = NULL;
5460011   msg.path_cwd_size = 0;
5460012   sys (SYS_UAREA, &msg, (sizeof msg));
5460013   return (msg.ppid);
5460014 }
```

95.30.27 lib/unistd/getuid.c

Si veda la sezione 87.27.

```
5470001 #include <unistd.h>
5470002 #include <sys/types.h>
5470003 #include <sys/os32.h>
5470004 #include <errno.h>
5470005 //-----------------------------------------------------------
5470006 uid_t
5470007 getuid (void)
5470008 {
5470009   sysmsg_uarea_t msg;
5470010   msg.path_cwd = NULL;
5470011   msg.path_cwd_size = 0;
5470012   sys (SYS_UAREA, &msg, (sizeof msg));
5470013   return (msg.uid);
5470014 }
```

95.30.28 lib/unistd/isatty.c

Si veda la sezione 88.69.

```
5480001 #include <sys/stat.h>
5480002 #include <sys/os32.h>
5480003 #include <unistd.h>
5480004 #include <sys/types.h>
5480005 #include <errno.h>
5480006 //-----------------------------------------------------------
5480007 int
5480008 isatty (int fdn)
5480009 {
5480010   struct stat file_status;
5480011   //
5480012   // Verify to have valid input data.
5480013   //
5480014   if (fdn < 0)
5480015     {
5480016       errset (EBADF);
5480017       return (0);
5480018     }
5480019   //
5480020   // Verify the standard input.
5480021   //
5480022   if (fstat (fdn, &file_status) == 0)
5480023     {
5480024       if (major (file_status.st_rdev) == DEV_CONSOLE_MAJOR)
5480025         {
5480026           return (1);   // Meaning it is ok!
5480027         }
5480028       if (major (file_status.st_rdev) == DEV_TTY_MAJOR)
5480029         {
5480030           return (1);   // Meaning it is ok!
5480031         }
5480032     }
5480033   else
5480034     {
5480035       errset (errno);
5480036       return (0);
5480037     }
5480038   //
5480039   // If here, it is not a terminal of any kind.
5480040   //
5480041   errset (EINVAL);
5480042   return (0);
5480043 }
```


95.30.29 lib/unistd/link.c

Si veda la sezione 87.30.

```
5490001 #include <unistd.h>
5490002 #include <string.h>
5490003 #include <sys/os32.h>
5490004 #include <errno.h>
5490005 #include <limits.h>
5490006 //-----------------------------------------------------------
5490007 int
5490008 link (const char *path_old, const char *path_new)
5490009 {
5490010   sysmsg_link_t msg;
5490011   //
5490012   msg.path_old = path_old;
5490013   msg.path_new = path_new;
5490014   //
5490015   sys (SYS_LINK, &msg, (sizeof msg));
5490016   //
5490017   errno = msg.errno;
5490018   errln = msg.errln;
5490019   strncpy (errfn, msg.errfn, PATH_MAX);
5490020   return (msg.ret);
5490021 }
```

95.30.30 lib/unistd/lseek.c

Si veda la sezione 87.33.

```
5500001 #include <unistd.h>
5500002 #include <sys/types.h>
5500003 #include <sys/os32.h>
5500004 #include <errno.h>
5500005 #include <string.h>
5500006 //-----------------------------------------------------------
5500007 off_t
5500008 lseek (int fdn, off_t offset, int whence)
5500009 {
5500010   sysmsg_lseek_t msg;
5500011   msg.fdn = fdn;
5500012   msg.offset = offset;
5500013   msg.whence = whence;
5500014   sys (SYS_LSEEK, &msg, (sizeof msg));
5500015   errno = msg.errno;
5500016   errln = msg.errln;
5500017   strncpy (errfn, msg.errfn, PATH_MAX);
5500018   return (msg.ret);
5500019 }
```

95.30.31 lib/unistd/pipe.c

Si veda la sezione 87.38.

```
5510001 #include <unistd.h>
5510002 #include <string.h>
5510003 #include <sys/os32.h>
5510004 #include <errno.h>
5510005 #include <limits.h>
5510006 //-----------------------------------------------------------
5510007 int
5510008 pipe (int pipefd[2])
5510009 {
5510010   sysmsg_pipe_t msg;
5510011   //
5510012   if (pipefd == NULL)
5510013     {
5510014       errset (EINVAL);
5510015       return (-1);
5510016     }
5510017   //
5510018   sys (SYS_PIPE, &msg, (sizeof msg));
5510019   //
5510020   errno = msg.errno;
5510021   errln = msg.errln;
5510022   //
5510023   pipefd[0] = msg.pipefd[0];
5510024   pipefd[1] = msg.pipefd[1];
5510025   //
5510026   return (msg.ret);
5510027 }
```

95.30.32 lib/unistd/read.c

Si veda la sezione 87.39.

```
5520001 #include <unistd.h>
5520002 #include <sys/os32.h>
5520003 #include <errno.h>
5520004 #include <string.h>
5520005 #include <stdio.h>
5520006 #include <fcntl.h>
5520007 //---------------------------------------------------------
5520008 ssize_t
5520009 read (int fdn, void *buffer, size_t count)
5520010 {
5520011   sysmsg_read_t msg;
5520012   //
5520013   // Reduce size of read if necessary.
5520014   //
5520015   if (count > BUFSIZ)
5520016     {
5520017       count = BUFSIZ;
5520018     }
5520019   //
5520020   // Fill the message.
5520021   //
5520022   msg.fdn = fdn;
5520023   msg.buffer = buffer;
5520024   msg.count = count;
5520025   msg.fl_flags = 0;     // Not necessary.
5520026   msg.ret = 0;
5520027   //
5520028   // Repeat syscall, until something is received or
5520029   // end of file is
5520030   // reached.
5520031   //
5520032   while (1)
5520033     {
5520034       sys (SYS_READ, &msg, (sizeof msg));
5520035       if (msg.ret == 0)
5520036         {
5520037           //
5520038           // End of file.
5520039           //
5520040           break;
5520041         }
5520042       if (msg.ret < 0
5520043           && (msg.errno == EAGAIN
5520044               || msg.errno == EWOULDBLOCK))
5520045         {
5520046           //
5520047           // No data at the moment.
5520048           //
5520049           if (msg.fl_flags & O_NONBLOCK)
5520050             {
5520051               //
5520052               // Don't block.
5520053               //
5520054               break;
5520055             }
5520056           else
5520057             {
5520058               //
5520059               // Keep trying.
5520060               //
5520061               continue;
5520062             }
5520063         }
5520064       //
5520065       // Otherwise, we have read something.
5520066       //
5520067       break;
5520068     }
5520069   //
5520070   //
5520071   //
5520072   if (msg.ret < 0)
5520073     {
5520074       //
5520075       // No valid read.
5520076       //
5520077       errno = msg.errno;
5520078       errln = msg.errln;
5520079       strncpy (errfn, msg.errfn, PATH_MAX);
5520080       return (msg.ret);
5520081     }
5520082   //
5520083   if (msg.ret > count)
5520084     {
5520085       //
5520086       // A strange value was returned. Considering it
5520087       // a read error.
5520088       //
5520089       errset (EIO);     // I/O error.
5520090       return (-1);
5520091     }
5520092   //
5520093   // A valid read: return.
5520094   //
5520095   return (msg.ret);
5520096 }
```


95.30.33 lib/unistd/rmdir.c

Si veda la sezione 87.41.

```
5530001 #include <unistd.h>
5530002 #include <string.h>
5530003 #include <sys/os32.h>
5530004 #include <errno.h>
5530005 #include <limits.h>
5530006 //---------------------------------------------------------
5530007 int
5530008 rmdir (const char *path)
5530009 {
5530010   sysmsg_stat_t msg_stat;
5530011   sysmsg_unlink_t msg_unlink;
5530012   //
5530013   msg_stat.path = path;
5530014   //
5530015   sys (SYS_STAT, &msg_stat, (sizeof msg_stat));
5530016   //
5530017   if (msg_stat.ret != 0)
5530018     {
5530019       errno = msg_stat.errno;
5530020       errln = msg_stat.errln;
5530021       strncpy (errfn, msg_stat.errfn, PATH_MAX);
5530022       return (msg_stat.ret);
5530023     }
5530024   //
5530025   if (!S_ISDIR (msg_stat.stat.st_mode))
5530026     {
5530027       errset (ENOTDIR); // Not a directory.
5530028       return (-1);
5530029     }
5530030   //
5530031   msg_unlink.path = path;
5530032   //
5530033   sys (SYS_UNLINK, &msg_unlink, (sizeof msg_unlink));
5530034   //
5530035   errno = msg_unlink.errno;
5530036   errln = msg_unlink.errln;
5530037   strncpy (errfn, msg_unlink.errfn, PATH_MAX);
5530038   return (msg_unlink.ret);
5530039 }
```

95.30.34 lib/unistd/sbrk.c

Si veda la sezione 87.5.

```
5540001 #include <unistd.h>
5540002 #include <string.h>
5540003 #include <sys/os32.h>
5540004 #include <errno.h>
5540005 #include <limits.h>
5540006 //---------------------------------------------------------
5540007 void *
5540008 sbrk (intptr_t increment)
5540009 {
5540010   sysmsg_sbrk_t msg_sbrk;
5540011   //
5540012   msg_sbrk.increment = increment;
5540013   //
5540014   sys (SYS_SBRK, &msg_sbrk, (sizeof msg_sbrk));
5540015   //
5540016   errno = msg_sbrk.errno;
5540017   errln = msg_sbrk.errln;
5540018   strncpy (errfn, msg_sbrk.errfn, PATH_MAX);
5540019   return (msg_sbrk.ret);
5540020 }
```

95.30.35 lib/unistd/setegid.c

Si veda la sezione 87.48.

```
5550001 #include <unistd.h>
5550002 #include <sys/types.h>
5550003 #include <sys/os32.h>
5550004 #include <errno.h>
5550005 #include <string.h>
5550006 //-----------------------------------------------------------
5550007 int
5550008 setegid (gid_t gid)
5550009 {
5550010   sysmsg_setegid_t msg;
5550011   msg.ret = 0;
5550012   msg.errno = 0;
5550013   msg.egid = gid;
5550014   sys (SYS_SETEGID, &msg, (sizeof msg));
5550015   errno = msg.errno;
5550016   errln = msg.errln;
5550017   strncpy (errfn, msg.errfn, PATH_MAX);
5550018   return (msg.ret);
5550019 }
```

95.30.36 lib/unistd/seteuid.c

Si veda la sezione 87.51.

```
5560001 #include <unistd.h>
5560002 #include <sys/types.h>
5560003 #include <sys/os32.h>
5560004 #include <errno.h>
5560005 #include <string.h>
5560006 //-----------------------------------------------------------
5560007 int
5560008 seteuid (uid_t uid)
5560009 {
5560010   sysmsg_seteuid_t msg;
5560011   msg.ret = 0;
5560012   msg.errno = 0;
5560013   msg.euid = uid;
5560014   sys (SYS_SETEGID, &msg, (sizeof msg));
5560015   errno = msg.errno;
5560016   errln = msg.errln;
5560017   strncpy (errfn, msg.errfn, PATH_MAX);
5560018   return (msg.ret);
5560019 }
```

95.30.37 lib/unistd/setgid.c

Si veda la sezione 87.48.

```
5570001 #include <unistd.h>
5570002 #include <sys/types.h>
5570003 #include <sys/os32.h>
5570004 #include <errno.h>
5570005 #include <string.h>
5570006 //-----------------------------------------------------------
5570007 int
5570008 setgid (gid_t gid)
5570009 {
5570010   sysmsg_setgid_t msg;
5570011   msg.ret = 0;
5570012   msg.errno = 0;
5570013   msg.egid = gid;
5570014   sys (SYS_SETGID, &msg, (sizeof msg));
5570015   errno = msg.errno;
5570016   errln = msg.errln;
5570017   strncpy (errfn, msg.errfn, PATH_MAX);
5570018   return (msg.ret);
5570019 }
```

95.30.38 lib/unistd/setpgrp.c

Si veda la sezione 87.50.

```
5580001 #include <unistd.h>
5580002 #include <sys/os32.h>
5580003 #include <stddef.h>
5580004 //-----------------------------------------------------------
5580005 int
5580006 setpgrp (void)
5580007 {
5580008   sys (SYS_PGRP, NULL, (size_t) 0);
5580009   return (0);
5580010 }
```


95.30.39 lib/unistd/setuid.c

Si veda la sezione 87.51.

```
5590001 #include <unistd.h>
5590002 #include <sys/types.h>
5590003 #include <sys/os32.h>
5590004 #include <errno.h>
5590005 #include <string.h>
5590006 //-----------------------------------------------------------
5590007 int
5590008 setuid (uid_t uid)
5590009 {
5590010   sysmsg_setuid_t msg;
5590011   msg.ret = 0;
5590012   msg.errno = 0;
5590013   msg.euid = uid;
5590014   sys (SYS_SETUID, &msg, (sizeof msg));
5590015   errno = msg.errno;
5590016   errln = msg.errln;
5590017   strncpy (errfn, msg.errfn, PATH_MAX);
5590018   return (msg.ret);
5590019 }
```

95.30.40 lib/unistd/sleep.c

Si veda la sezione 87.53.

```
5600001 #include <unistd.h>
5600002 #include <sys/types.h>
5600003 #include <sys/os32.h>
5600004 #include <errno.h>
5600005 #include <time.h>
5600006 //-----------------------------------------------------------
5600007 unsigned int
5600008 sleep (unsigned int seconds)
5600009 {
5600010   sysmsg_sleep_t msg;
5600011   time_t start;
5600012   time_t end;
5600013   int slept;
5600014   //
5600015   if (seconds == 0)
5600016     {
5600017       return (0);
5600018     }
5600019   //
5600020   msg.events = WAKEUP_EVENT_TIMER;
5600021   msg.seconds = seconds;
5600022   sys (SYS_SLEEP, &msg, (sizeof msg));
5600023   start = msg.ret;
5600024   end = time (NULL);
5600025   slept = end - msg.ret;
5600026   //
5600027   if (slept < 0)
5600028     {
5600029       return (seconds);
5600030     }
5600031   else if (slept < seconds)
5600032     {
5600033       return (seconds - slept);
5600034     }
5600035   else
5600036     {
5600037       return (0);
5600038     }
5600039 }
```

95.30.41 lib/unistd/ttyname.c

Si veda la sezione 88.133.

```
5610001 #include <sys/os32.h>
5610002 #include <sys/stat.h>
5610003 #include <unistd.h>
5610004 #include <sys/types.h>
5610005 #include <errno.h>
5610006 #include <limits.h>
5610007 //-----------------------------------------------------------
5610008 char *
5610009 ttyname (int fdn)
5610010 {
5610011   dev_t dev_minor;
5610012   struct stat file_status;
5610013   static char name[PATH_MAX];
5610014   //
5610015   // Verify to have valid input data.
5610016   //
```

```
5610017   if (fdn < 0)
5610018     {
5610019       errset (EBADF);
5610020       return (NULL);
5610021     }
5610022   //
5610023   // Verify the file descriptor.
5610024   //
5610025   if (fstat (fdn, &file_status) == 0)
5610026     {
5610027       if (major (file_status.st_rdev) == DEV_CONSOLE_MAJOR)
5610028         {
5610029           dev_minor = minor (file_status.st_rdev);
5610030           //
5610031           // If minor is equal to 0xFF, it is
5610032           // '/dev/console'.
5610033           //
5610034           if (dev_minor < 0xFF)
5610035             {
5610036               sprintf (name, "/dev/console%i", dev_minor);
5610037             }
5610038           else
5610039             {
5610040               strcpy (name, "/dev/console");
5610041             }
5610042           return (name);
5610043         }
5610044       else if (file_status.st_rdev == DEV_TTY)
5610045         {
5610046           strcpy (name, "/dev/tty");
5610047           return (name);
5610048         }
5610049       else
5610050         {
5610051           errset (ENOTTY);
5610052           return (NULL);
5610053         }
5610054     }
5610055   else
5610056     {
5610057       errset (errno);
5610058       return (NULL);
5610059     }
5610060 }
```

95.30.42 lib/unistd/unlink.c

«

Si veda la sezione 87.62.

```
5620001 #include <unistd.h>
5620002 #include <string.h>
5620003 #include <sys/os32.h>
5620004 #include <errno.h>
5620005 #include <limits.h>
5620006 //-------------------------------------------------------------
5620007 int
5620008 unlink (const char *path)
5620009 {
5620010   sysmsg_unlink_t msg;
5620011   //
5620012   msg.path = path;
5620013   //
5620014   sys (SYS_UNLINK, &msg, (sizeof msg));
5620015   //
5620016   errno = msg.errno;
5620017   errln = msg.errln;
5620018   strncpy (errfn, msg.errfn, PATH_MAX);
5620019   return (msg.ret);
5620020 }
```

95.30.43 lib/unistd/write.c

«

Si veda la sezione 87.64.

```
5630001 #include <unistd.h>
5630002 #include <sys/os32.h>
5630003 #include <errno.h>
5630004 #include <string.h>
5630005 #include <stdio.h>
5630006 //-------------------------------------------------------------
5630007 ssize_t
5630008 write (int fdn, const void *buffer, size_t count)
5630009 {
5630010   sysmsg_write_t msg;
5630011   //
5630012   // Reduce size of write if necessary.
5630013   //
```



```
5630014   if (count > BUFSIZ)
5630015     {
5630016       count = BUFSIZ;
5630017     }
5630018   //
5630019   // Fill the message.
5630020   //
5630021   msg.fdn = fdn;
5630022   msg.buffer = buffer;
5630023   msg.count = count;
5630024   //
5630025   // Syscall.
5630026   //
5630027   sys (SYS_WRITE, &msg, (sizeof msg));
5630028   //
5630029   // Check result and return.
5630030   //
5630031   if (msg.ret < 0)
5630032     {
5630033       //
5630034       // No valid write.
5630035       //
5630036       errno = msg.errno;
5630037       errln = msg.errln;
5630038       strncpy (errfn, msg.errfn, PATH_MAX);
5630039       return (msg.ret);
5630040     }
5630041   //
5630042   if (msg.ret > count)
5630043     {
5630044       //
5630045       // A strange value was returned. Considering it
5630046       // a read error.
5630047       //
5630048       errset (EIO);     // I/O error.
5630049       return (-1);
5630050     }
5630051   //
5630052   // A valid write return.
5630053   //
5630054   return (msg.ret);
5630055 }
```

95.31 os32: «lib/utime.h»

«

Si veda la sezione 91.3.

```
5640001 #ifndef _UTIME_H
5640002 #define _UTIME_H        1
5640003 //-------------------------------------------------------------
5640004 #include <restrict.h>
5640005 #include <sys/types.h>  // time_t
5640006 //-------------------------------------------------------------
5640007 struct utimbuf
5640008 {
5640009   time_t actime;
5640010   time_t modtime;
5640011 };
5640012 //-------------------------------------------------------------
5640013 int utime (const char *path, const struct utimbuf *times);
5640014 //-------------------------------------------------------------
5640015
5640016 #endif
```

95.31.1 lib/utime/utime.c 935

95.31.1 lib/utime/utime.c

«

Si veda la sezione 91.3.

```
5650001 #include <utime.h>
5650002 #include <errno.h>
5650003 //-------------------------------------------------------------
5650004 int
5650005 utime (const char *path, const struct utimbuf *times)
5650006 {
5650007   //
5650008   // Currently not implemented.
5650009   //
5650010   return (0);
5650011 }
```