Sorgenti della libreria generale

«

«a2» 2013.11.11 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://informaticalibera.net

os16: file isolati della directory «lib/»

«

lib/NULL.h

«

Si veda la sezione u0.2.

```
2030001 #ifndef _NULL_H
2030002 #define _NULL_H      1
2030003
2030004 #define NULL 0
2030005
2030006 #endif
```

lib/SEEK.h

«

Si veda la sezione u0.2.

```
2040001  #ifndef _SEEK_H
2040002  #define _SEEK_H      1
2040003
2040004  //------------------------------------------------------------------------
2040005  // These values are used inside <stdio.h> and <unistd.h>
2040006  //------------------------------------------------------------------------
2040007  #define SEEK_SET        0  // From the start.
2040008  #define SEEK_CUR        1  // From current position.
2040009  #define SEEK_END        2  // From the end.
2040010  //------------------------------------------------------------------------
2040011
2040012  #endif
```

lib/_Bool.h

«

Si veda la sezione u0.2.

```
2050001  #ifndef __BOOL_H
2050002  #define __BOOL_H      1
2050003
2050004  typedef unsigned char _Bool;
2050005
2050006  #endif
```

lib/clock_t.h

«

Si veda la sezione u0.2.

```
2060001  #ifndef _CLOCK_T_H
2060002  #define _CLOCK_T_H       1
2060003
2060004  typedef unsigned long int clock_t;      // 32 bit unsigned int.
2060005
2060006  #endif
```

lib/const.h

«

Si veda la sezione u0.2.

```
2070001  #ifndef _CONST_H
2070002  #define _CONST_H        1
2070003
2070004  #define const
2070005
2070006  #endif
```

lib/ctype.h

«

Si veda la sezione u0.2.

```
2080001  #ifndef _CTYPE_H
2080002  #define _CTYPE_H       1
2080003  //---------------------------------------------------------------------
2080004
2080005  #include <NULL.h>
2080006  //---------------------------------------------------------------------
2080007  #define isblank(C)   ((int) (C == ' ' || C == '\t'))
2080008  #define isspace(C)   ((int) (C == ' ' \
2080009                               || C == '\f' \
2080010                               || C == '\n' \
2080011                               || C == '\n' \
2080012                               || C == '\r' \
2080013                               || C == '\t' \
2080014                               || C == '\v'))
2080015  #define isdigit(C)   ((int) (C >= '0' && C <= '9'))
2080016  #define isxdigit(C) ((int) ((C >= '0' && C <= '9') \
2080017                                || (C >= 'A' && C <= 'F') \
2080018                                || (C >= 'a' && C <= 'f')))
2080019  #define isupper(C)   ((int) (C >= 'A' && C <= 'Z'))
2080020  #define islower(C)   ((int) (C >= 'a' && C <= 'z'))
2080021  #define iscntrl(C)   ((int) ((C >= 0x00 && C <= 0x1F) || C == 0x7F))
2080022  #define isgraph(C)   ((int) (C >= 0x21 && C <= 0x7E))
2080023  #define isprint(C)   ((int) (C >= 0x20 && C <= 0x7E))
2080024  #define isalpha(C)   (isupper (C) || islower (C))
2080025  #define isalnum(C)   (isalpha (C) || isdigit (C))
2080026  #define ispunct(C)   (isgraph (C) && (!isspace (C)) && (!isalnum (C)))
2080027  #define tolower(C)   (isupper (C) ? ((C) + 0x20) : (C))
2080028  #define toupper(C)   (islower (C) ? ((C) - 0x20) : (C))
```

```
2080029 #define toascii(C)   (C & 0x7F)
2080030 #define _tolower(C) (isupper (C) ? ((C) + 0x20) : (C))
2080031 #define _toupper(C) (islower (C) ? ((C) - 0x20) : (C))
2080032 //------------------------------------------------------------------------
2080033
2080034 #endif
```

lib/inttypes.h

Si veda la sezione u0.2.

```
2090001 #ifndef _INTTYPES_H
2090002 #define _INTTYPES_H     1
2090003 //------------------------------------------------------------------------
2090004
2090005 #include <const.h>
2090006 #include <restrict.h>
2090007 #include <stdint.h>
2090008 #include <wchar_t.h>
2090009 //------------------------------------------------------------------------
2090010 typedef struct {
2090011     intmax_t quot;
2090012     intmax_t rem;
2090013 } imaxdiv_t;
2090014 //
2090015 imaxdiv_t imaxdiv    (intmax_t numer, intmax_t denom);
2090016 //------------------------------------------------------------------------
2090017 // Output typesetting.
2090018 //------------------------------------------------------------------------
2090019 #define PRId8           "d"
2090020 #define PRId16          "d"
2090021 #define PRId32          "ld"
2090022 #define PRIdLEAST8      "d"
2090023 #define PRIdLEAST16     "d"
2090024 #define PRIdLEAST32     "ld"
2090025 #define PRIdFAST8       "d"
2090026 #define PRIdFAST16      "d"
2090027 #define PRIdFAST32      "ld"
2090028 #define PRIdMAX         "ld"
2090029 #define PRIdPTR         "d"
2090030 #define PRIi8           "i"
2090031 #define PRIi16          "i"
```

```
2090032 #define PRIi32          "li"
2090033 #define PRIiLEAST8      "i"
2090034 #define PRIiLEAST16     "i"
2090035 #define PRIiLEAST32     "li"
2090036 #define PRIiFAST8       "i"
2090037 #define PRIiFAST16      "i"
2090038 #define PRIiFAST32      "i"
2090039 #define PRIiMAX         "li"
2090040 #define PRIiPTR         "i"
2090041 #define PRIo8           "o"
2090042 #define PRIo16          "o"
2090043 #define PRIo32          "lo"
2090044 #define PRIoLEAST8      "o"
2090045 #define PRIoLEAST16     "o"
2090046 #define PRIoLEAST32     "lo"
2090047 #define PRIoFAST8       "o"
2090048 #define PRIoFAST16      "o"
2090049 #define PRIoFAST32      "lo"
2090050 #define PRIoMAX         "lo"
2090051 #define PRIoPTR         "o"
2090052 #define PRIu8           "u"
2090053 #define PRIu16          "u"
2090054 #define PRIu32          "lu"
2090055 #define PRIuLEAST8      "u"
2090056 #define PRIuLEAST16     "u"
2090057 #define PRIuLEAST32     "lu"
2090058 #define PRIuFAST8       "u"
2090059 #define PRIuFAST16      "u"
2090060 #define PRIuFAST32      "lu"
2090061 #define PRIuMAX         "lu"
2090062 #define PRIuPTR         "u"
2090063 #define PRIx8           "x"
2090064 #define PRIx16          "x"
2090065 #define PRIx32          "lx"
2090066 #define PRIxLEAST8      "x"
2090067 #define PRIxLEAST16     "x"
2090068 #define PRIxLEAST32     "lx"
2090069 #define PRIxFAST8       "x"
2090070 #define PRIxFAST16      "x"
2090071 #define PRIxFAST32      "lx"
2090072 #define PRIxMAX         "lx"
2090073 #define PRIxPTR         "x"
2090074 #define PRIX8           "X"
```

```
2090075 #define PRIX16          "X"
2090076 #define PRIX32          "lX"
2090077 #define PRIXLEAST8      "X"
2090078 #define PRIXLEAST16     "X"
2090079 #define PRIXLEAST32     "lX"
2090080 #define PRIXFAST8       "X"
2090081 #define PRIXFAST16      "X"
2090082 #define PRIXFAST32      "lX"
2090083 #define PRIXMAX         "lX"
2090084 #define PRIXPTR         "X"
2090085 //------------------------------------------------------------------------
2090086 // Input scan and evaluation.
2090087 //------------------------------------------------------------------------
2090088 #define SCNd8           "hhd"
2090089 #define SCNd16          "hd"
2090090 #define SCNd32          "d"
2090091 #define SCNdLEAST8      "hhd"
2090092 #define SCNdLEAST16     "hd"
2090093 #define SCNdLEAST32     "d"
2090094 #define SCNdFAST8       "hhd"
2090095 #define SCNdFAST16      "d"
2090096 #define SCNdFAST32      "d"
2090097 #define SCNdMAX         "ld"
2090098 #define SCNdPTR         "d"
2090099 #define SCNi8           "hhi"
2090100 #define SCNi16          "hi"
2090101 #define SCNi32          "i"
2090102 #define SCNiLEAST8      "hhi"
2090103 #define SCNiLEAST16     "hi"
2090104 #define SCNiLEAST32     "i"
2090105 #define SCNiFAST8       "hhi"
2090106 #define SCNiFAST16      "i"
2090107 #define SCNiFAST32      "i"
2090108 #define SCNiMAX         "li"
2090109 #define SCNiPTR         "i"
2090110 #define SCNo8           "hho"
2090111 #define SCNo16          "ho"
2090112 #define SCNo32          "o"
2090113 #define SCNoLEAST8      "hho"
2090114 #define SCNoLEAST16     "ho"
2090115 #define SCNoLEAST32     "o"
2090116 #define SCNoFAST8       "hho"
2090117 #define SCNoFAST16      "o"
```

```
2090118 #define SCNoFAST32      "o"
2090119 #define SCNoMAX         "lo"
2090120 #define SCNoPTR         "o"
2090121 #define SCNu8           "hhu"
2090122 #define SCNu16          "hu"
2090123 #define SCNu32          "u"
2090124 #define SCNuLEAST8      "hhu"
2090125 #define SCNuLEAST16     "hu"
2090126 #define SCNuLEAST32     "u"
2090127 #define SCNuFAST8       "hhu"
2090128 #define SCNuFAST16      "u"
2090129 #define SCNuFAST32      "u"
2090130 #define SCNuMAX         "lu"
2090131 #define SCNuPTR         "u"
2090132 #define SCNx8           "hhx"
2090133 #define SCNx16          "hx"
2090134 #define SCNx32          "x"
2090135 #define SCNxLEAST8      "hhx"
2090136 #define SCNxLEAST16     "hx"
2090137 #define SCNxLEAST32     "x"
2090138 #define SCNxFAST8       "hhx"
2090139 #define SCNxFAST16      "x"
2090140 #define SCNxFAST32      "x"
2090141 #define SCNxMAX         "lx"
2090142 #define SCNxPTR         "x"
2090143 //-------------------------------------------------------------------
2090144 intmax_t  strtoimax  (const char *restrict nptr,
2090145                       char **restrict endptr, int base);
2090146 uintmax_t strtouimax (const char *restrict nptr,
2090147                       char **restrict endptr, int base);
2090148 intmax_t  wcstoimax  (const wchar_t *restrict nptr,
2090149                       wchar_t **restrict endptr, int base);
2090150 uintmax_t wcstouimax (const wchar_t *restrict nptr,
2090151                       wchar_t **restrict endptr, int base);
2090152 //-------------------------------------------------------------------
2090153
2090154 #endif
```

«

Si veda la sezione u0.2.

```
2100001 #ifndef _LIMITS_H
2100002 #define _LIMITS_H       1
2100003 //------------------------------------------------------------------------
2100004 #define CHAR_BIT               (8)
2100005 #define SCHAR_MIN           (-0x80)
2100006 #define SCHAR_MAX            (0x7F)
2100007 #define UCHAR_MAX            (0xFF)
2100008 #define CHAR_MIN         SCHAR_MIN
2100009 #define CHAR_MAX         SCHAR_MAX
2100010 #define MB_LEN_MAX             (1)
2100011 #define SHRT_MIN          (-0x8000)
2100012 #define SHRT_MAX           (0x7FFF)
2100013 #define USHRT_MAX          (0xFFFF)
2100014 #define INT_MIN           (-0x8000)
2100015 #define INT_MAX            (0x7FFF)
2100016 #define UINT_MAX           (0xFFFFU)
2100017 #define LONG_MIN      (-0x80000000L)
2100018 #define LONG_MAX       (0x7FFFFFFFL)
2100019 #define ULONG_MAX      (0xFFFFFFFFUL)
2100020 //------------------------------------------------------------------------
2100021 #define LLONG_MIN     (-0x80000000L)      // The type 'long long int'
2100022 #define LLONG_MAX      (0x7FFFFFFFL)      // is not available with
2100023 #define ULLONG_MAX     (0xFFFFFFFFUL)     // a K&R C compiler.
2100024 //------------------------------------------------------------------------
2100025 #define WORD_BIT               16  // POSIX requires at least 32!
2100026 #define LONG_BIT               32
2100027 #define SSIZE_MAX         LONG_MAX
2100028 //------------------------------------------------------------------------
2100029 #define ARG_MAX              1024  // Arguments+environment max length.
2100030 #define ATEXIT_MAX             32  // Max "at exit" functions.
2100031 #define FILESIZEBITS           32  // File size needs integer size...
2100032 #define LINK_MAX              254  // Max links per file.
2100033 #define NAME_MAX               14  // File name max (Minix 1 fs).
2100034 #define OPEN_MAX                8  // Max open files per process.
2100035 #define PATH_MAX               64  // Path, including final '\0'.
2100036 #define MAX_CANON               1  // Max bytes in canonical tty queue.
2100037 #define MAX_INPUT               1  // Max bytes in tty input queue.
2100038 //------------------------------------------------------------------------
2100039 #define CHLD_MAX           INT_MAX  // Not used.
2100040 #define HOST_NAME_MAX      INT_MAX  // Not used.
```

```
2100041  #define LOGIN_NAME_MAX     INT_MAX  // Not used.
2100042  #define PAGE_SIZE          INT_MAX  // Not used.
2100043  #define RE_DUP_MAX         INT_MAX  // Not used.
2100044  #define STREAM_MAX         INT_MAX  // Not used.
2100045  #define SYMLOOP_MAX        INT_MAX  // Not used.
2100046  #define TTY_NAME_MAX       INT_MAX  // Not used.
2100047  #define TZNAME_MAX         INT_MAX  // Not used.
2100048  #define PIPE_MAX           INT_MAX  // Not used.
2100049  #define SYMLINK_MAX        INT_MAX  // Not used.
2100050  //------------------------------------------------------------------------
2100051
2100052  #endif
```

lib/ptrdiff_t.h

«

Si veda la sezione u0.2.

```
2110001  #ifndef _PTRDIFF_T_H
2110002  #define _PTRDIFF_T_H    1
2110003
2110004  typedef int ptrdiff_t;
2110005
2110006  #endif
```

lib/restrict.h

«

Si veda la sezione u0.2.

```
2120001  #ifndef _RESTRICT_H
2120002  #define _RESTRICT_H      1
2120003
2120004  #define restrict
2120005
2120006  #endif
```

lib/size_t.h

«

Si veda la sezione u0.2.

```
2130001  #ifndef _SIZE_T_H
2130002  #define _SIZE_T_H       1
2130003  //
2130004  // The type 'size_t' *must* be equal to an 'int'.
2130005  //
2130006  typedef unsigned int size_t;
2130007
2130008  #endif
```

lib/stdarg.h

«

Si veda la sezione u0.2.

```
2140001  #ifndef _STDARG_H
2140002  #define _STDARG_H       1
2140003  //---------------------------------------------------------------------
2140004  typedef unsigned char *va_list;
2140005  //---------------------------------------------------------------------
2140006
2140007  #define va_start(ap, last) ((void) ((ap) = \
2140008                              ((va_list) &(last)) + (sizeof (last))))
2140009  #define va_end(ap)         ((void) ((ap) = 0))
2140010  #define va_copy(dest, src) ((void) ((dest) = (va_list) (src)))
2140011  #define va_arg(ap, type)   (((ap) = (ap) + (sizeof (type))), \
2140012                              *((type *) ((ap) - (sizeof (type)))))
2140013  //---------------------------------------------------------------------
2140014
2140015  #endif
```

lib/stdbool.h

«

Si veda la sezione u0.2.

```
2150001  #ifndef _STDBOOL_H
2150002  #define _STDBOOL_H      1
2150003  //---------------------------------------------------------------------
2150004  typedef unsigned char bool;     // [1]
2150005  #define true   ((bool) 1)
```

```
2150006 #define false  ((bool) 0)
2150007 #define __bool_true_false_are_defined   1
2150008 //
2150009 // [1] For some reason, it cannot be defined as a macro expanding to
2150010 //      '_Bool'. Anyway, it is the same kind of type.
2150011 //
2150012 //---------------------------------------------------------------------
2150013
2150014 #endif
```

lib/stddef.h

«

Si veda la sezione u0.2.

```
2160001 #ifndef _STDDEF_H
2160002 #define _STDDEF_H      1
2160003 //---------------------------------------------------------------------
2160004
2160005 #include <ptrdiff_t.h>
2160006 #include <size_t.h>
2160007 #include <wchar_t.h>
2160008 #include <NULL.h>
2160009 //---------------------------------------------------------------------
2160010 #define offsetof(type, member)  ((size_t) &((type *)0)->member)
2160011 //---------------------------------------------------------------------
2160012
2160013 #endif
```

lib/stdint.h

«

Si veda la sezione u0.2.

```
2170001 #ifndef _STDINT_H
2170002 #define _STDINT_H       1
2170003 //---------------------------------------------------------------------
2170004 typedef signed char                  int8_t;
2170005 typedef short int                   int16_t;
2170006 typedef long int                    int32_t;
2170007 typedef unsigned char              uint8_t;
2170008 typedef unsigned short int         uint16_t;
2170009 typedef unsigned long int          uint32_t;
```

```
2170010 //
2170011 #define INT8_MIN                    (-0x80)
2170012 #define INT8_MAX                     (0x7F)
2170013 #define UINT8_MAX                    (0xFF)
2170014 #define INT16_MIN                 (-0x8000)
2170015 #define INT16_MAX                  (0x7FFF)
2170016 #define UINT16_MAX                 (0xFFFF)
2170017 #define INT32_MIN              (-0x80000000)
2170018 #define INT32_MAX               (0x7FFFFFFF)
2170019 #define UINT32_MAX              (0xFFFFFFFFU)
2170020 //------------------------------------------------------------------------
2170021 typedef signed char            int_least8_t;
2170022 typedef short int             int_least16_t;
2170023 typedef long int              int_least32_t;
2170024 typedef unsigned char         uint_least8_t;
2170025 typedef unsigned short int   uint_least16_t;
2170026 typedef unsigned long int    uint_least32_t;
2170027 //
2170028 #define INT_LEAST8_MIN              (-0x80)
2170029 #define INT_LEAST8_MAX               (0x7F)
2170030 #define UINT_LEAST8_MAX              (0xFF)
2170031 #define INT_LEAST16_MIN           (-0x8000)
2170032 #define INT_LEAST16_MAX            (0x7FFF)
2170033 #define UINT_LEAST16_MAX           (0xFFFF)
2170034 #define INT_LEAST32_MIN        (-0x80000000)
2170035 #define INT_LEAST32_MAX         (0x7FFFFFFF)
2170036 #define UINT_LEAST32_MAX        (0xFFFFFFFFU)
2170037 //------------------------------------------------------------------------
2170038 #define INT8_C(VAL)                 VAL
2170039 #define INT16_C(VAL)                VAL
2170040 #define INT32_C(VAL)                VAL
2170041 #define UINT8_C(VAL)                VAL
2170042 #define UINT16_C(VAL)               VAL
2170043 #define UINT32_C(VAL)               VAL ## U
2170044 //------------------------------------------------------------------------
2170045 typedef signed char             int_fast8_t;
2170046 typedef int                    int_fast16_t;
2170047 typedef long int               int_fast32_t;
2170048 typedef unsigned char          uint_fast8_t;
2170049 typedef unsigned int           uint_fast16_t;
2170050 typedef unsigned long int      uint_fast32_t;
2170051 //
2170052 #define INT_FAST8_MIN               (-0x80)
```

```
2170053 #define INT_FAST8_MAX                  (0x7F)
2170054 #define UINT_FAST8_MAX                 (0xFF)
2170055 #define INT_FAST16_MIN           (-0x80000000)
2170056 #define INT_FAST16_MAX            (0x7FFFFFFF)
2170057 #define UINT_FAST16_MAX           (0xFFFFFFFFU)
2170058 #define INT_FAST32_MIN           (-0x80000000)
2170059 #define INT_FAST32_MAX            (0x7FFFFFFF)
2170060 #define UINT_FAST32_MAX           (0xFFFFFFFFU)
2170061 //------------------------------------------------------------------------
2170062 typedef int                           intptr_t;
2170063 typedef unsigned int                 uintptr_t;
2170064 //
2170065 #define INTPTR_MIN               (-0x80000000)
2170066 #define INTPTR_MAX               (0x7FFFFFFF)
2170067 #define UINTPTR_MAX              (0xFFFFFFFFU)
2170068 //------------------------------------------------------------------------
2170069 typedef long int                     intmax_t;
2170070 typedef unsigned long int           uintmax_t;
2170071 //
2170072 #define INTMAX_C(VAL)              VAL ## L
2170073 #define UINTMAX_C(VAL)             VAL ## UL
2170074 //
2170075 #define INTMAX_MIN              (-0x80000000L)
2170076 #define INTMAX_MAX               (0x7FFFFFFFL)
2170077 #define UINTMAX_MAX              (0xFFFFFFFFUL)
2170078 //------------------------------------------------------------------------
2170079 #define PTRDIFF_MIN             (-0x80000000)
2170080 #define PTRDIFF_MAX              (0x7FFFFFFF)
2170081 //
2170082 #define SIG_ATOMIC_MIN          (-0x80000000)
2170083 #define SIG_ATOMIC_MAX           (0x7FFFFFFF)
2170084 //
2170085 #define SIZE_MAX                      (0xFFFFU)
2170086 //
2170087 #define WCHAR_MIN                        (0)
2170088 #define WCHAR_MAX                     (0xFFU)
2170089 //
2170090 #define WINT_MIN                     (-0x80L)
2170091 #define WINT_MAX                      (0x7FL)
2170092 //------------------------------------------------------------------------
2170093
2170094 #endif
```

lib/time_t.h

«

Si veda la sezione u0.2.

```
2180001  #ifndef _TIME_T_H
2180002  #define _TIME_T_H       1
2180003
2180004  typedef long int time_t;
2180005
2180006  #endif
```

lib/wchar_t.h

«

Si veda la sezione u0.2.

```
2190001  #ifndef _WCHAR_T_H
2190002  #define _WCHAR_T_H      1
2190003
2190004  typedef unsigned char wchar_t;
2190005
2190006  #endif
```

os16: «lib/dirent.h»

«

Si veda la sezione u0.2.

```
2200001  #ifndef _DIRENT_H
2200002  #define _DIRENT_H        1
2200003
2200004  #include <sys/types.h>  // ino_t
2200005  #include <limits.h>     // NAME_MAX
2200006  #include <const.h>
2200007
2200008  //------------------------------------------------------------------------
2200009  struct dirent {
2200010      ino_t  d_ino;                   // I-node number [1]
2200011      char   d_name[NAME_MAX+1];      // NAME_MAX + Null termination
2200012  };
2200013  //
2200014  // [1] The type 'ino_t' must be equal to 'uint16_t', because the
2200015  //     directory inside the Minix 1 file system has exactly such
```

```
2200016 //     size.
2200017 //
2200018 //------------------------------------------------------------------------
2200019 #define DOPEN_MAX   OPEN_MAX/2  // <limits.h> [1]
2200020 //
2200021 // [1] DOPEN_MAX is not standard, but it is used to define how many
2200022 //     directory slot to keep for open directories. As directory streams
2200023 //     are opened as file descriptors, the sum of all kind of file open
2200024 //     cannot be more than OPEM_MAX.
2200025 //------------------------------------------------------------------------
2200026 typedef struct {
2200027     int           fdn;       // File descriptor number.
2200028     struct dirent dir;       // Last directory item read.
2200029 } DIR;
2200030
2200031 extern DIR _directory_stream[]; // Defined inside 'lib/dirent/DIR.c'.
2200032 //------------------------------------------------------------------------
2200033 // Function prototypes.
2200034 //------------------------------------------------------------------------
2200035 int            closedir  (DIR *dp);
2200036 DIR           *opendir   (const char *name);
2200037 struct dirent *readdir   (DIR *dp);
2200038 void           rewinddir (DIR *dp);
2200039 //------------------------------------------------------------------------
2200040
2200041 #endif
```

lib/dirent/DIR.c

«

Si veda la sezione u0.2.

```
2210001 #include <dirent.h>
2210002 //
2210003 // There must be room for at least 'DOPEN_MAX' elements.
2210004 //
2210005 DIR _directory_stream[DOPEN_MAX];
2210006
2210007 void
2210008 _dirent_directory_stream_setup (void)
2210009 {
2210010     int d;
2210011     //
```

```
2210012       for (d = 0; d < DOPEN_MAX; d++)
2210013         {
2210014           _directory_stream[d].fdn   = -1;
2210015         }
2210016   }
```

lib/dirent/closedir.c

«

Si veda la sezione u0.10.

```
2220001   #include <dirent.h>
2220002   #include <fcntl.h>
2220003   #include <const.h>
2220004   #include <sys/types.h>
2220005   #include <sys/stat.h>
2220006   #include <unistd.h>
2220007   #include <errno.h>
2220008   #include <stddef.h>
2220009   //------------------------------------------------------------------------
2220010   int
2220011   closedir (DIR *dp)
2220012   {
2220013       //
2220014       // Check for a valid argument
2220015       //
2220016       if (dp == NULL)
2220017         {
2220018           //
2220019           // Not a valid pointer.
2220020           //
2220021           errset (EBADF);          // Invalid directory.
2220022           return (-1);
2220023         }
2220024       //
2220025       // Check if it is an open directory stream.
2220026       //
2220027       if (dp->fdn < 0)
2220028         {
2220029           //
2220030           // The stream is closed.
2220031           //
2220032           errset (EBADF);          // Invalid directory.
```

```
2220033           return (-1);
2220034         }
2220035       //
2220036       // Close the file descriptor. It there is an error,
2220037       // the 'errno' variable will be set by 'close()'.
2220038       //
2220039       return (close (dp->fdn));
2220040   }
```

lib/dirent/opendir.c

«

Si veda la sezione u0.76.

```
2230001   #include <dirent.h>
2230002   #include <fcntl.h>
2230003   #include <const.h>
2230004   #include <sys/types.h>
2230005   #include <sys/stat.h>
2230006   #include <unistd.h>
2230007   #include <errno.h>
2230008   #include <stddef.h>
2230009   //---------------------------------------------------------------------
2230010   DIR *
2230011   opendir (const char *path)
2230012   {
2230013       int    fdn;
2230014       int    d;
2230015       DIR  *dp;
2230016       struct stat file_status;
2230017       //
2230018       // Function 'opendir()' is used only for reading.
2230019       //
2230020       fdn = open (path, O_RDONLY);
2230021       //
2230022       // Check the file descriptor returned.
2230023       //
2230024       if (fdn < 0)
2230025         {
2230026           //
2230027           // The variable 'errno' is already set:
2230028           //   EINVAL
2230029           //   EMFILE
```

```
2230030           //   ENFILE
2230031           //
2230032           errset (errno);
2230033           return (NULL);
2230034         }
2230035       //
2230036       // Set the 'FD_CLOEXEC' flag for that file descriptor.
2230037       //
2230038       if (fcntl (fdn, F_SETFD, FD_CLOEXEC) != 0)
2230039         {
2230040           //
2230041           // The variable 'errno' is already set:
2230042           //   EBADF
2230043           //
2230044           errset (errno);
2230045           close (fdn);
2230046           return (NULL);
2230047         }
2230048       //
2230049       //
2230050       //
2230051       if (fstat (fdn, &file_status) != 0)
2230052         {
2230053           //
2230054           // Error should be already set.
2230055           //
2230056           errset (errno);
2230057           close (fdn);
2230058           return (NULL);
2230059         }
2230060       //
2230061       // Verify it is a directory
2230062       //
2230063       if (!S_ISDIR(file_status.st_mode))
2230064         {
2230065           //
2230066           // It is not a directory!
2230067           //
2230068           close (fdn);
2230069           errset (ENOTDIR);               // Is not a directory.
2230070           return (NULL);
2230071         }
2230072       //
```

```
2230073      // A valid file descriptor is available: must find a free
2230074      // `_directory_stream[]' slot.
2230075      //
2230076      for (d = 0; d < DOPEN_MAX; d++)
2230077        {
2230078          if (_directory_stream[d].fdn < 0)
2230079            {
2230080              //
2230081              // Found a free slot: set it up.
2230082              //
2230083              dp = &(_directory_stream[d]);
2230084              dp->fdn   = fdn;
2230085              //
2230086              // Return the directory pointer.
2230087              //
2230088              return (dp);
2230089            }
2230090        }
2230091      //
2230092      // If we are here, there was no free directory slot available.
2230093      //
2230094      close  (fdn);
2230095      errset (EMFILE);                    // Too many file open.
2230096      return (NULL);
2230097  }
```

lib/dirent/readdir.c

«

Si veda la sezione u0.86.

```
2240001  #include <dirent.h>
2240002  #include <fcntl.h>
2240003  #include <sys/types.h>
2240004  #include <sys/stat.h>
2240005  #include <unistd.h>
2240006  #include <errno.h>
2240007  #include <stddef.h>
2240008  //---------------------------------------------------------------------
2240009  struct dirent *
2240010  readdir (DIR *dp)
2240011  {
2240012      ssize_t size;
```

```
2240013       //
2240014       // Check for a valid argument.
2240015       //
2240016       if (dp == NULL)
2240017         {
2240018           //
2240019           // Not a valid pointer.
2240020           //
2240021           errset (EBADF);         // Invalid directory.
2240022           return (NULL);
2240023         }
2240024       //
2240025       // Check if it is an open directory stream.
2240026       //
2240027       if (dp->fdn < 0)
2240028         {
2240029           //
2240030           // The stream is closed.
2240031           //
2240032           errset (EBADF);         // Invalid directory.
2240033           return (NULL);
2240034         }
2240035       //
2240036       // Read the directory.
2240037       //
2240038       size = read (dp->fdn, &(dp->dir),
2240039                    (size_t) 16);
2240040       //
2240041       // Fix the null termination, if the name is very long.
2240042       //
2240043       dp->dir.d_name[NAME_MAX] = '\0';
2240044       //
2240045       // Check what was read.
2240046       //
2240047       if (size == 0)
2240048         {
2240049           //
2240050           // End of directory, but it is not an error.
2240051           //
2240052           return (NULL);
2240053         }
2240054       //
2240055       if (size < 0)
```

```
2240056         {
2240057           //
2240058           // This is an error. The variable ‘errno’ is already set.
2240059           //
2240060           errset (errno);
2240061           return (NULL);
2240062         }
2240063       //
2240064       if (dp->dir.d_ino == 0)
2240065         {
2240066           //
2240067           // This is a null directory record.
2240068           // Should try to read the next one.
2240069           //
2240070           return (readdir (dp));
2240071         }
2240072       //
2240073       if (strlen (dp->dir.d_name) == 0)
2240074         {
2240075           //
2240076           // This is a bad directory record: try to read next.
2240077           //
2240078           return (readdir (dp));
2240079         }
2240080       //
2240081       // A valid directory record should be available now.
2240082       //
2240083       return (&(dp->dir));
2240084   }
```

lib/dirent/rewinddir.c

«

Si veda la sezione u0.89.

```
2250001  #include <dirent.h>
2250002  #include <fcntl.h>
2250003  #include <const.h>
2250004  #include <sys/types.h>
2250005  #include <sys/stat.h>
2250006  #include <unistd.h>
2250007  #include <errno.h>
2250008  #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);
}
```

os16: «lib/errno.h»

«

Si veda la sezione u0.18.

```
#ifndef _ERRNO_H
#define _ERRNO_H       1

```

```
#include <limits.h>
#include <string.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 macro-instruction: '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 available on the stream head
                           // read queue.
#define ENODEV          44 // No such device.
#define ENOENT          45 // No such file or directory.
#define ENOEXEC         46 // Executable file format error.
#define ENOLCK          47 // No locks available.
#define ENOLINK         48 // Reserved.
#define ENOMEM          49 // Not enough space.
#define ENOMSG          50 // No message of the desired type.
#define ENOPROTOOPT     51 // Protocol not available.
#define ENOSPC          52 // No space left on device.
#define ENOSR           53 // No stream resources.
#define ENOSTR          54 // Not a stream.
#define ENOSYS          55 // Function not supported.
#define ENOTCONN        56 // The socket is not connected.
#define ENOTDIR         57 // Not a directory.
#define ENOTEMPTY       58 // Directory not empty.
#define ENOTSOCK        59 // Not a socket.
#define ENOTSUP         60 // Not supported.
#define ENOTTY          61 // Inappropriate I/O control operation.
#define ENXIO           62 // No such device or address.
#define EOPNOTSUPP      63 // Operation not supported on socket.
#define EOVERFLOW       64 // Value too large to be stored in data type.
#define EPERM           65 // Operation not permitted.
#define EPIPE           66 // Broken pipe.
```

```
#define EPROTO          67 // Protocol error.
#define EPROTONOSUPPORT 68 // Protocol not supported.
#define EPROTOTYPE      69 // Protocol wrong type for socket.
#define ERANGE          70 // Result too large.
#define EROFS           71 // Read-only file system.
#define ESPIPE          72 // Invalid seek.
#define ESRCH           73 // No such process.
#define ESTALE          74 // Reserved.
#define ETIME           75 // Stream ioctl() timeout.
#define ETIMEDOUT       76 // Connection timed out.
#define ETXTBSY         77 // Text file busy.
#define EWOULDBLOCK     78 // Operation would block
                           // (may be the same as EAGAIN).
#define EXDEV           79 // Cross-device link.
//--------------------------------------------------------------------------
// Added os16 errors.
//--------------------------------------------------------------------------
#define EUNKNOWN                   (-1) // Unknown error.
#define E_FILE_TYPE                 80  // File type not compatible.
#define E_ROOT_INODE_NOT_CACHED     81  // The root directory inode is
                                        // not cached.
#define E_CANNOT_READ_SUPERBLOCK    83  // Cannot read super block.
#define E_MAP_INODE_TOO_BIG         84  // Map inode too big.
#define E_MAP_ZONE_TOO_BIG          85  // Map zone too big.
#define E_DATA_ZONE_TOO_BIG         86  // Data zone too big.
#define E_CANNOT_FIND_ROOT_DEVICE   87  // Cannot find root device.
#define E_CANNOT_FIND_ROOT_INODE    88  // Cannot find root inode.
#define E_FILE_TYPE_UNSUPPORTED     89  // File type unsupported.
#define E_ENV_TOO_BIG               90  // Environment too big.
#define E_LIMIT                     91  // Exceeded implementation
                                        // limits.
#define E_NOT_MOUNTED               92  // Not mounted.
#define E_NOT_IMPLEMENTED           93  // Not implemented.
//--------------------------------------------------------------------------
// Default descriptions for errors.
//--------------------------------------------------------------------------
#define TEXT_E2BIG                      "Argument list too long."
#define TEXT_EACCES                     "Permission denied."
#define TEXT_EADDRINUSE                 "Address in use."
#define TEXT_EADDRNOTAVAIL              "Address not available."
#define TEXT_EAFNOSUPPORT               "Address family not supported."
#define TEXT_EAGAIN                     "Resource unavailable, " \
                                        "try again."
```

```
2260133 #define TEXT_EALREADY                   "Connection already in " \
2260134                                         "progress."
2260135 #define TEXT_EBADF                      "Bad file descriptor."
2260136 #define TEXT_EBADMSG                    "Bad message."
2260137 #define TEXT_EBUSY                      "Device or resource busy."
2260138 #define TEXT_ECANCELED                  "Operation canceled."
2260139 #define TEXT_ECHILD                     "No child processes."
2260140 #define TEXT_ECONNABORTED               "Connection aborted."
2260141 #define TEXT_ECONNREFUSED               "Connection refused."
2260142 #define TEXT_ECONNRESET                 "Connection reset."
2260143 #define TEXT_EDEADLK                    "Resource deadlock would occur."
2260144 #define TEXT_EDESTADDRREQ               "Destination address required."
2260145 #define TEXT_EDOM                       "Mathematics argument out of " \
2260146                                         "domain of function."
2260147 #define TEXT_EDQUOT                     "Reserved error: EDQUOT"
2260148 #define TEXT_EEXIST                     "File exists."
2260149 #define TEXT_EFAULT                     "Bad address."
2260150 #define TEXT_EFBIG                      "File too large."
2260151 #define TEXT_EHOSTUNREACH               "Host is unreachable."
2260152 #define TEXT_EIDRM                      "Identifier removed."
2260153 #define TEXT_EILSEQ                     "Illegal byte sequence."
2260154 #define TEXT_EINPROGRESS                "Operation in progress."
2260155 #define TEXT_EINTR                      "Interrupted function."
2260156 #define TEXT_EINVAL                     "Invalid argument."
2260157 #define TEXT_EIO                        "I/O error."
2260158 #define TEXT_EISCONN                    "Socket is connected."
2260159 #define TEXT_EISDIR                     "Is a directory."
2260160 #define TEXT_ELOOP                      "Too many levels of " \
2260161                                         "symbolic links."
2260162 #define TEXT_EMFILE                     "Too many open files."
2260163 #define TEXT_EMLINK                     "Too many links."
2260164 #define TEXT_EMSGSIZE                   "Message too large."
2260165 #define TEXT_EMULTIHOP                  "Reserved error: EMULTIHOP"
2260166 #define TEXT_ENAMETOOLONG               "Filename too long."
2260167 #define TEXT_ENETDOWN                   "Network is down."
2260168 #define TEXT_ENETRESET                  "Connection aborted by network."
2260169 #define TEXT_ENETUNREACH                "Network unreachable."
2260170 #define TEXT_ENFILE                     "Too many files open in system."
2260171 #define TEXT_ENOBUFS                    "No buffer space available."
2260172 #define TEXT_ENODATA                    "No message is available on " \
2260173                                         "the stream head read queue."
2260174 #define TEXT_ENODEV                     "No such device."
2260175 #define TEXT_ENOENT                     "No such file or directory."
```

```
2260176 #define TEXT_ENOEXEC                    "Executable file format error."
2260177 #define TEXT_ENOLCK                     "No locks available."
2260178 #define TEXT_ENOLINK                    "Reserved error: ENOLINK"
2260179 #define TEXT_ENOMEM                     "Not enough space."
2260180 #define TEXT_ENOMSG                     "No message of the desired " \
2260181                                         "type."
2260182 #define TEXT_ENOPROTOOPT                "Protocol not available."
2260183 #define TEXT_ENOSPC                     "No space left on device."
2260184 #define TEXT_ENOSR                      "No stream resources."
2260185 #define TEXT_ENOSTR                     "Not a stream."
2260186 #define TEXT_ENOSYS                     "Function not supported."
2260187 #define TEXT_ENOTCONN                   "The socket is not connected."
2260188 #define TEXT_ENOTDIR                    "Not a directory."
2260189 #define TEXT_ENOTEMPTY                  "Directory not empty."
2260190 #define TEXT_ENOTSOCK                   "Not a socket."
2260191 #define TEXT_ENOTSUP                    "Not supported."
2260192 #define TEXT_ENOTTY                     "Inappropriate I/O control " \
2260193                                         "operation."
2260194 #define TEXT_ENXIO                      "No such device or address."
2260195 #define TEXT_EOPNOTSUPP                 "Operation not supported on " \
2260196                                         "socket."
2260197 #define TEXT_EOVERFLOW                  "Value too large to be " \
2260198                                         "stored in data type."
2260199 #define TEXT_EPERM                      "Operation not permitted."
2260200 #define TEXT_EPIPE                      "Broken pipe."
2260201 #define TEXT_EPROTO                     "Protocol error."
2260202 #define TEXT_EPROTONOSUPPORT            "Protocol not supported."
2260203 #define TEXT_EPROTOTYPE                 "Protocol wrong type for " \
2260204                                         "socket."
2260205 #define TEXT_ERANGE                     "Result too large."
2260206 #define TEXT_EROFS                      "Read-only file system."
2260207 #define TEXT_ESPIPE                     "Invalid seek."
2260208 #define TEXT_ESRCH                      "No such process."
2260209 #define TEXT_ESTALE                     "Reserved error: ESTALE"
2260210 #define TEXT_ETIME                      "Stream ioctl() timeout."
2260211 #define TEXT_ETIMEDOUT                  "Connection timed out."
2260212 #define TEXT_ETXTBSY                    "Text file busy."
2260213 #define TEXT_EWOULDBLOCK                "Operation would block."
2260214 #define TEXT_EXDEV                      "Cross-device link."
2260215 //----------------------------------------------------------------------
2260216 #define TEXT_EUNKNOWN                   "Unknown error."
2260217 #define TEXT_E_FILE_TYPE                "File type not compatible."
2260218 #define TEXT_E_ROOT_INODE_NOT_CACHED    "The root directory inode " \
```

```
2260219                                         "is not cached."
2260220 #define TEXT_E_CANNOT_READ_SUPERBLOCK   "Cannot read super block."
2260221 #define TEXT_E_MAP_INODE_TOO_BIG        "Map inode too big."
2260222 #define TEXT_E_MAP_ZONE_TOO_BIG         "Map zone too big."
2260223 #define TEXT_E_DATA_ZONE_TOO_BIG        "Data zone too big."
2260224 #define TEXT_E_CANNOT_FIND_ROOT_DEVICE  "Cannot find root device."
2260225 #define TEXT_E_CANNOT_FIND_ROOT_INODE   "Cannot find root inode."
2260226 #define TEXT_E_FILE_TYPE_UNSUPPORTED    "File type unsupported."
2260227 #define TEXT_E_ENV_TOO_BIG              "Environment too big."
2260228 #define TEXT_E_LIMIT                    "Exceeded implementation " \
2260229                                         "limits."
2260230 #define TEXT_E_NOT_MOUNTED               "Not mounted."
2260231 #define TEXT_E_NOT_IMPLEMENTED          "Not implemented."
2260232
2260233 //-------------------------------------------------------------------------
2260234 // The function 'error()' is not standard and is used to return a
2260235 // pointer to a string containing the default description of the
2260236 // error contained inside 'errno'.
2260237 //-------------------------------------------------------------------------
2260238 char *error      (void);        // Not standard!
2260239
2260240 #endif
```

lib/errno/errno.c

«

Si veda la sezione u0.18.

```
2270001 //-------------------------------------------------------------------------
2270002 // This file does not include the 'errno.h' header, because here 'errno'
2270003 // should not be declared as an extern variable!
2270004 //-------------------------------------------------------------------------
2270005
2270006 #include <limits.h>
2270007 //-------------------------------------------------------------------------
2270008 // The variable 'errno' is standard, but 'errln' and 'errfn' are added
2270009 // to keep track of the error source. Variable 'errln' is used to save
2270010 // the source file line number; variable 'errfn' is used to save the
2270011 // source file name. To set these variable in a consistent way it is
2270012 // also added a macro-instruction: 'errset'.
2270013 //-------------------------------------------------------------------------
2270014 int  errno;
2270015 int  errln;
```

```
2270016 char errfn[PATH_MAX];
2270017 //-----------------------------------------------------------------------
```

os16: «lib/fcntl.h»

«

Si veda la sezione u0.2.

```
2280001 #ifndef _FCNTL_H
2280002 #define _FCNTL_H        1
2280003
2280004 #include <const.h>
2280005 #include <sys/types.h>  // mode_t
2280006                         // off_t
2280007                         // pid_t
2280008 //-----------------------------------------------------------------------
2280009 // Values for the second parameter of function 'fcntl()'.
2280010 //-----------------------------------------------------------------------
2280011 #define F_DUPFD         0       // Duplicate file descriptor.
2280012 #define F_GETFD         1       // Get file descriptor flags.
2280013 #define F_SETFD         2       // Set file descriptor flags.
2280014 #define F_GETFL         3       // Get file status flags.
2280015 #define F_SETFL         4       // Set file status flags.
2280016 #define F_GETLK         5       // Get record locking information.
2280017 #define F_SETLK         6       // Set record locking information.
2280018 #define F_SETLKW        7       // Set record locking information;
2280019                                 // wait if blocked.
2280020 #define F_GETOWN        8       // Set owner of socket.
2280021 #define F_SETOWN        9       // Get owner of socket.
2280022 //-----------------------------------------------------------------------
2280023 // Flags to be set with:
2280024 //   fcntl (fd, F_SETFD, ...);
2280025 //-----------------------------------------------------------------------
2280026 #define FD_CLOEXEC      1       // Close the file descriptor upon
2280027                                 // execution of an exec() family
2280028                                 // function.
2280029 //-----------------------------------------------------------------------
2280030 // Values for type 'l_type', used for record locking with 'fcntl()'.
2280031 //-----------------------------------------------------------------------
2280032 #define F_RDLCK         0       // Read lock.
2280033 #define F_WRLCK         1       // Write lock.
2280034 #define F_UNLCK         2       // Remove lock.
2280035 //-----------------------------------------------------------------------
```

```
// Flags for file creation, in place of 'oflag' parameter for function
// 'open()'.
//--------------------------------------------------------------------
#define O_CREAT         000010  // Create file if it does not exist.
#define O_EXCL          000020  // Exclusive use flag.
#define O_NOCTTY        000040  // Do not assign a controlling terminal.
#define O_TRUNC         000100  // Truncation flag.
//--------------------------------------------------------------------
// Flags for the file status, used with 'open()' and 'fcntl()'.
//--------------------------------------------------------------------
#define O_APPEND        000200  // Write append.
#define O_DSYNC         000400  // Synchronized write operations.
#define O_NONBLOCK      001000  // Non-blocking mode.
#define O_RSYNC         002000  // Synchronized read operations.
#define O_SYNC          004000  // Synchronized read and write.
//--------------------------------------------------------------------
// File access mask selection.
//--------------------------------------------------------------------
#define O_ACCMODE       000003  // Mask to select the last three bits,
                                // used to specify the main access
                                // modes: read, write and both.
//--------------------------------------------------------------------
// Main access modes.
//--------------------------------------------------------------------
#define O_RDONLY        000001                  // Read.
#define O_WRONLY        000002                  // Write.
#define O_RDWR          (O_RDONLY | O_WRONLY)   // Both read and write.
//--------------------------------------------------------------------
// Structure 'flock', used to file lock for POSIX standard. It is not
// used inside os16.
//--------------------------------------------------------------------
struct flock {
    short int l_type;   // Type of lock: F_RDLCK, F_WRLCK, or F_UNLCK.
    short int l_whence; // Start reference point.
    off_t     l_start;  // Offset, from 'l_whence', for the area start.
    off_t     l_len;    // Locked area size. Zero means up to the end of
                        // the file.
    pid_t     l_pid;    // The process id blocking the area.
};
//--------------------------------------------------------------------
// Function prototypes.
//--------------------------------------------------------------------
int creat (const char *path, mode_t mode);
```

```
2280079  int fcntl (int fdn, int cmd, ...);
2280080  int open  (const char *path, int oflags, ...);
2280081  //-------------------------------------------------------------------
2280082
2280083  #endif
```

lib/fcntl/creat.c

«

Si veda la sezione u0.11.

```
2290001  #include <fcntl.h>
2290002  #include <sys/types.h>
2290003  #include <const.h>
2290004  //-------------------------------------------------------------------
2290005  int
2290006  creat (const char *path, mode_t mode)
2290007  {
2290008      return (open (path, O_WRONLY|O_CREAT|O_TRUNC, mode));
2290009  }
```

lib/fcntl/fcntl.c

«

Si veda la sezione u0.13.

```
2300001  #include <fcntl.h>
2300002  #include <stdarg.h>
2300003  #include <stddef.h>
2300004  #include <string.h>
2300005  #include <errno.h>
2300006  #include <sys/os16.h>
2300007  #include <const.h>
2300008  #include <limits.h>
2300009  //-------------------------------------------------------------------
2300010  int
2300011  fcntl (int fdn, int cmd, ...)
2300012  {
2300013      va_list ap;
2300014      sysmsg_fcntl_t msg;
2300015      va_start (ap, cmd);
2300016      //
2300017      // Well known arguments.
```

```
2300018       //
2300019       msg.fdn = fdn;
2300020       msg.cmd = cmd;
2300021       //
2300022       // Select other arguments.
2300023       //
2300024       switch (cmd)
2300025         {
2300026           case F_DUPFD:
2300027           case F_SETFD:
2300028           case F_SETFL:
2300029               msg.arg = va_arg (ap, int);
2300030               break;
2300031           case F_GETFD:
2300032           case F_GETFL:
2300033               break;
2300034           case F_GETOWN:
2300035           case F_SETOWN:
2300036           case F_GETLK:
2300037           case F_SETLK:
2300038           case F_SETLKW:
2300039               errset (E_NOT_IMPLEMENTED);          // Not implemented.
2300040               return (-1);
2300041           default:
2300042               errset (EINVAL);                     // Not implemented.
2300043               return (NULL);
2300044         }
2300045       //
2300046       // Do the system call.
2300047       //
2300048       sys (SYS_FCNTL, &msg, (sizeof msg));
2300049       errno = msg.errno;
2300050       errln = msg.errln;
2300051       strncpy (errfn, msg.errfn, PATH_MAX);
2300052       return (msg.ret);
2300053   }
```

lib/fcntl/open.c

«

Si veda la sezione u0.28.

```
2310001  #include <fcntl.h>
2310002  #include <stdarg.h>
2310003  #include <stddef.h>
2310004  #include <string.h>
2310005  #include <errno.h>
2310006  #include <sys/os16.h>
2310007  #include <const.h>
2310008  #include <limits.h>
2310009  //------------------------------------------------------------------------
2310010  int
2310011  open (const char *path, int oflags, ...)
2310012  {
2310013      va_list ap;
2310014      sysmsg_open_t msg;
2310015      va_start (ap, oflags);
2310016      if (path == NULL || strlen (path) == 0)
2310017        {
2310018          errset (EINVAL);         // Invalid argument.
2310019          return (-1);
2310020        }
2310021      strncpy (msg.path, path, PATH_MAX);
2310022      msg.flags = oflags;
2310023      msg.mode  = va_arg (ap, mode_t);
2310024      sys (SYS_OPEN, &msg, (sizeof msg));
2310025      errno = msg.errno;
2310026      errln = msg.errln;
2310027      strncpy (errfn, msg.errfn, PATH_MAX);
2310028      return (msg.ret);
2310029  }
```

os16: «lib/grp.h»

«

Si veda la sezione u0.2.

```
2320001  //------------------------------------------------------------------------
2320002  // os16 does not have a group management!
2320003  //------------------------------------------------------------------------
2320004
2320005  #ifndef _GRP_H
```

```
2320006 #define _GRP_H        1
2320007
2320008 #include <const.h>
2320009 #include <restrict.h>
2320010 #include <sys/types.h>          // gid_t
2320011
2320012 //-----------------------------------------------------------------------
2320013 struct group {
2320014     char   *gr_name;
2320015     gid_t   gr_gid;
2320016     char  **gr_mem;
2320017 };
2320018 //-----------------------------------------------------------------------
2320019 struct group *getgrgid (gid_t gid);
2320020 struct group *getgrnam (const char *name);
2320021 //-----------------------------------------------------------------------
2320022
2320023 #endif
```

lib/grp/getgrgid.c

«

Si veda la sezione u0.2.

```
2330001 #include <grp.h>
2330002 #include <NULL.h>
2330003 //-----------------------------------------------------------------------
2330004 struct group *
2330005 getgrgid (gid_t gid)
2330006 {
2330007     static char *name = "none";
2330008     static struct group grp;
2330009     //
2330010     // os16 does not have a group management, so the answare is always
2330011     // the same.
2330012     //
2330013     grp.gr_name = name;
2330014     grp.gr_gid  = (gid_t) -1;
2330015     grp.gr_mem  = NULL;
2330016     //
2330017     return (&grp);
2330018 }
```

lib/grp/getgrnam.c

«

Si veda la sezione u0.2.

```
2340001  #include <grp.h>
2340002  //---------------------------------------------------------------------
2340003  struct group *
2340004  getgrnam (const char *name)
2340005  {
2340006      return (getgrgid ((gid_t) 0));
2340007  }
```

os16: «lib/libgen.h»

«

Si veda la sezione u0.2.

```
2350001  #ifndef _LIBGEN_H
2350002  #define _LIBGEN_H       1
2350003
2350004  //---------------------------------------------------------------------
2350005  char *basename (char *path);
2350006  char *dirname  (char *path);
2350007  //---------------------------------------------------------------------
2350008
2350009  #endif
```

lib/libgen/basename.c

«

Si veda la sezione u0.7.

```
2360001  #include <libgen.h>
2360002  #include <limits.h>
2360003  #include <stddef.h>
2360004  #include <string.h>
2360005  //---------------------------------------------------------------------
2360006  char *
2360007  basename (char *path)
2360008  {
2360009      static char  *point = ".";          // When 'path' is NULL.
2360010             char  *p;                    // Pointer inside 'path'.
2360011             int    i;                    // Scan index inside 'path'.
```

```
2360012       //
2360013       // Empty path.
2360014       //
2360015       if (path == NULL || strlen (path) == 0)
2360016         {
2360017           return (point);
2360018         }
2360019       //
2360020       // Remove all final `/' if it exists, excluded the first character:
2360021       // `i' is kept greater than zero.
2360022       //
2360023       for (i = (strlen (path) - 1); i > 0 && path[i] == '/'; i--)
2360024         {
2360025           path[i] = 0;
2360026         }
2360027       //
2360028       // After removal of extra final `/', if there is only one `/', this
2360029       // is to be returned.
2360030       //
2360031       if (strncmp (path, "/",  PATH_MAX) == 0)
2360032         {
2360033           return (path);
2360034         }
2360035       //
2360036       // If there are no `/'.
2360037       //
2360038       if (strchr (path, '/') == NULL)
2360039         {
2360040           return (path);
2360041         }
2360042       //
2360043       // Find the last `/' and calculate a pointer to the base name.
2360044       //
2360045       p = strrchr (path, (unsigned int) '/');
2360046       p++;
2360047       //
2360048       // Return the pointer to the base name.
2360049       //
2360050       return (p);
2360051   }
```

«

Si veda la sezione u0.7.

```
2370001 #include <libgen.h>
2370002 #include <limits.h>
2370003 #include <stddef.h>
2370004 #include <string.h>
2370005 //------------------------------------------------------------------
2370006 char *
2370007 dirname (char *path)
2370008 {
2370009     static char  *point = ".";          // When 'path' is NULL.
2370010            char  *p;                    // Pointer inside 'path'.
2370011            int    i;                    // Scan index inside 'path'.
2370012     //
2370013     // Empty path.
2370014     //
2370015     if      (path == NULL || strlen (path) == 0)
2370016       {
2370017         return (point);
2370018       }
2370019     //
2370020     // Simple cases.
2370021     //
2370022     if (strncmp (path, "/",  PATH_MAX) == 0  ||
2370023         strncmp (path, ".",  PATH_MAX) == 0  ||
2370024         strncmp (path, "..", PATH_MAX) == 0)
2370025       {
2370026         return (path);
2370027       }
2370028     //
2370029     // Remove all final '/' if it exists, excluded the first character:
2370030     // 'i' is kept greater than zero.
2370031     //
2370032     for (i = (strlen (path) - 1); i > 0 && path[i] == '/'; i--)
2370033       {
2370034         path[i] = 0;
2370035       }
2370036     //
2370037     // After removal of extra final '/', if there is only one '/', this
2370038     // is to be returned.
2370039     //
2370040     if (strncmp (path, "/",  PATH_MAX) == 0)
```

```
2370041         {
2370042           return (path);
2370043         }
2370044       //
2370045       // If there are no `/'
2370046       //
2370047       if (strchr (path, '/') == NULL)
2370048         {
2370049           return (point);
2370050         }
2370051       //
2370052       // If there is only a `/' a the beginning.
2370053       //
2370054       if (path[0] == '/'                                    &&
2370055           strchr (&path[1], (unsigned int) '/') == NULL)
2370056         {
2370057           path[1] = 0;
2370058           return (path);
2370059         }
2370060       //
2370061       // Replace the last `/' with zero.
2370062       //
2370063        p = strrchr (path, (unsigned int) '/');
2370064       *p = 0;
2370065       //
2370066       // Now remove extra duplicated final `/', except the very first
2370067       // character: `i' is kept greater than zero.
2370068       //
2370069       for (i = (strlen (path) - 1); i > 0 && path[i] == '/'; i--)
2370070         {
2370071           path[i] = 0;
2370072         }
2370073       //
2370074       // Now `path' appears as a reduced string: the original path string
2370075       // is modified.
2370076       //
2370077       return (path);
2370078   }
```

os16: «lib/pwd.h»

«

Si veda la sezione u0.2.

```
2380001  #ifndef _PWD_H
2380002  #define _PWD_H        1
2380003
2380004  #include <const.h>
2380005  #include <restrict.h>
2380006  #include <sys/types.h>          // gid_t, uid_t
2380007  //---------------------------------------------------------------------
2380008  struct passwd {
2380009      char  *pw_name;
2380010      char  *pw_passwd;
2380011      uid_t  pw_uid;
2380012      gid_t  pw_gid;
2380013      char  *pw_gecos;
2380014      char  *pw_dir;
2380015      char  *pw_shell;
2380016  };
2380017  //---------------------------------------------------------------------
2380018  struct passwd *getpwent (void);
2380019  void           setpwent (void);
2380020  void           endpwent (void);
2380021  struct passwd *getpwnam (const char *name);
2380022  struct passwd *getpwuid (uid_t uid);
2380023  //---------------------------------------------------------------------
2380024
2380025  #endif
```

lib/pwd/pwent.c

«

Si veda la sezione u0.53.

```
2390001  #include <pwd.h>
2390002  #include <stdio.h>
2390003  #include <string.h>
2390004  #include <stdlib.h>
2390005
2390006  //---------------------------------------------------------------------
2390007  static char          buffer[BUFSIZ];
2390008  static struct passwd pw;
2390009  static FILE         *fp = NULL;
```

```
2390010 //---------------------------------------------------------------------
2390011 struct passwd *
2390012 getpwent (void)
2390013 {
2390014     void *pstatus;
2390015     char *char_uid;
2390016     char *char_gid;
2390017     //
2390018     if (fp == NULL)
2390019       {
2390020         fp = fopen ("/etc/passwd", "r");
2390021         if (fp == NULL)
2390022           {
2390023             return NULL;
2390024           }
2390025       }
2390026     //
2390027     pstatus = fgets (buffer, BUFSIZ, fp);
2390028     if (pstatus == NULL)
2390029       {
2390030         return (NULL);
2390031       }
2390032     //
2390033     pw.pw_name   = strtok (buffer, ":");
2390034     pw.pw_passwd = strtok (NULL, ":");
2390035     char_uid     = strtok (NULL, ":");
2390036     char_gid     = strtok (NULL, ":");
2390037     pw.pw_gecos  = strtok (NULL, ":");
2390038     pw.pw_dir    = strtok (NULL, ":");
2390039     pw.pw_shell  = strtok (NULL, ":");
2390040     pw.pw_uid    = (uid_t) atoi (char_uid);
2390041     pw.pw_gid    = (gid_t) atoi (char_gid);
2390042     //
2390043     return (&pw);
2390044 }
2390045 //---------------------------------------------------------------------
2390046 void
2390047 endpwent (void)
2390048 {
2390049     int status;
2390050     //
2390051     if (fp != NULL)
2390052       {
```

```
2390053         fclose (fp);
2390054         if (status != NULL)
2390055           {
2390056             fp = NULL;
2390057           }
2390058       }
2390059 }
2390060 //---------------------------------------------------------------------
2390061 void
2390062 setpwent (void)
2390063 {
2390064     if (fp != NULL)
2390065       {
2390066         rewind (fp);
2390067       }
2390068 }
2390069 //---------------------------------------------------------------------
2390070 struct passwd *
2390071 getpwnam (const char *name)
2390072 {
2390073     struct passwd *pw;
2390074     //
2390075     setpwent ();
2390076     //
2390077     for (;;)
2390078       {
2390079         pw = getpwent ();
2390080         if (pw == NULL)
2390081           {
2390082             return (NULL);
2390083           }
2390084         if (strcmp (pw->pw_name, name) == 0)
2390085           {
2390086             return (pw);
2390087           }
2390088       }
2390089 }
2390090 //---------------------------------------------------------------------
2390091 struct passwd *
2390092 getpwuid (uid_t uid)
2390093 {
2390094     struct passwd *pw;
2390095     //
```

```
2390096       setpwent ();
2390097       //
2390098       for (;;)
2390099         {
2390100           pw = getpwent ();
2390101           if (pw == NULL)
2390102             {
2390103               return (NULL);
2390104             }
2390105           if (pw->pw_uid == uid)
2390106             {
2390107               return (pw);
2390108             }
2390109         }
2390110   }
```

os16: «lib/signal.h»

«

Si veda la sezione u0.2.

```
2400001   #ifndef _SIGNAL_H
2400002   #define _SIGNAL_H       1
2400003
2400004   #include <sys/types.h>
2400005   //---------------------------------------------------------------------------
2400006   #define SIGHUP           1
2400007   #define SIGINT           2
2400008   #define SIGQUIT          3
2400009   #define SIGILL           4
2400010   #define SIGABRT          6
2400011   #define SIGFPE           8
2400012   #define SIGKILL          9
2400013   #define SIGSEGV         11
2400014   #define SIGPIPE         13
2400015   #define SIGALRM         14
2400016   #define SIGTERM         15
2400017   #define SIGSTOP         17
2400018   #define SIGTSTP         18
2400019   #define SIGCONT         19
2400020   #define SIGCHLD         20
2400021   #define SIGTTIN         21
2400022   #define SIGTTOU         22
```

```
2400023 #define SIGUSR1         30
2400024 #define SIGUSR2         31
2400025 //-------------------------------------------------------------------------
2400026 typedef int sig_atomic_t;
2400027 typedef void (*sighandler_t) (int); // The type 'sighandler_t' is a
2400028                                     // pointer to a function for the
2400029                                     // signal handling, with a parameter
2400030                                     // of type 'int', returning 'void'.
2400031 //
2400032 // Special undeclarable functions.
2400033 //
2400034 #define SIG_ERR ((sighandler_t) -1) // It transform an integer number
2400035 #define SIG_DFL ((sighandler_t) 0)  // into a 'sighandler_t' type,
2400036 #define SIG_IGN ((sighandler_t) 1)  // that is, a pointer to a function
2400037                                     // that does not exists really.
2400038 //-------------------------------------------------------------------------
2400039 sighandler_t signal (int sig, sighandler_t handler);
2400040 int          kill   (pid_t pid, int sig);
2400041 int          raise  (int sig);
2400042 //-------------------------------------------------------------------------
2400043
2400044 #endif
```

lib/signal/kill.c

«

Si veda la sezione u0.22.

```
2410001 #include <sys/os16.h>
2410002 #include <sys/types.h>
2410003 #include <signal.h>
2410004 #include <errno.h>
2410005 #include <string.h>
2410006 //-------------------------------------------------------------------------
2410007 int
2410008 kill (pid_t pid, int sig)
2410009 {
2410010     sysmsg_kill_t msg;
2410011     if (pid < -1)       // Currently unsupported.
2410012       {
2410013         errset (ESRCH);
2410014         return (-1);
2410015       }
```

```
2410016      msg.pid    = pid;
2410017      msg.signal = sig;
2410018      msg.ret    = 0;
2410019      msg.errno  = 0;
2410020      sys (SYS_KILL, &msg, (sizeof msg));
2410021      errno = msg.errno;
2410022      errln = msg.errln;
2410023      strncpy (errfn, msg.errfn, PATH_MAX);
2410024      return (msg.ret);
2410025  }
```

lib/signal/signal.c

«

Si veda la sezione u0.34.

```
2420001  #include <sys/os16.h>
2420002  #include <sys/types.h>
2420003  #include <signal.h>
2420004  #include <errno.h>
2420005  #include <string.h>
2420006  //------------------------------------------------------------------------
2420007  sighandler_t
2420008  signal (int sig, sighandler_t handler)
2420009  {
2420010      sysmsg_signal_t msg;
2420011
2420012      msg.signal  = sig;
2420013      msg.handler = handler;
2420014      msg.ret     = SIG_DFL;
2420015      msg.errno   = 0;
2420016      sys (SYS_SIGNAL, &msg, (sizeof msg));
2420017      errno = msg.errno;
2420018      errln = msg.errln;
2420019      strncpy (errfn, msg.errfn, PATH_MAX);
2420020      return (msg.ret);
2420021  }
```

os16: «lib/stdio.h»

«

Si veda la sezione u0.103.

```
2430001  #ifndef _STDIO_H
2430002  #define _STDIO_H        1
2430003
2430004  #include <const.h>
2430005  #include <restrict.h>
2430006  #include <stdarg.h>
2430007  #include <stdint.h>
2430008  #include <limits.h>
2430009  #include <NULL.h>
2430010  #include <size_t.h>
2430011  #include <sys/types.h>
2430012  #include <SEEK.h>       // SEEK_CUR, SEEK_SET, SEEK_END
2430013  //------------------------------------------------------------------------
2430014  #define BUFSIZ              2048  // Like the file system max zone
2430015                                    // size.
2430016  #define _IOFBF                 0  // Input-output fully buffered.
2430017  #define _IOLBF                 1  // Input-output line buffered.
2430018  #define _IONBF                 2  // Input-output with no buffering.
2430019
2430020  #define L_tmpnam    FILENAME_MAX  // <limits.h>
2430021
2430022  #define FOPEN_MAX       OPEN_MAX  // <limits.h>
2430023  #define FILENAME_MAX    NAME_MAX  // <limits.h>
2430024  #define TMP_MAX           0x7FFF
2430025
2430026  #define EOF                (-1)  // Must be a negative value.
2430027  //------------------------------------------------------------------------
2430028  typedef off_t     fpos_t;   // 'off_t' defined in <sys/types.h>.
2430029
2430030  typedef struct {
2430031      int           fdn;      // File descriptor number.
2430032      char          error;    // Error indicator.
2430033      char          eof;      // End of file indicator.
2430034  } FILE;
2430035
2430036  extern FILE _stream[];      // Defined inside 'lib/stdio/FILE.c'.
2430037
2430038  #define stdin   (&_stream[0])
2430039  #define stdout  (&_stream[1])
2430040  #define stderr  (&_stream[2])
```

```
2430041 //---------------------------------------------------------------------
2430042 void     clearerr   (FILE *fp);
2430043 int      fclose     (FILE *fp);
2430044 int      feof       (FILE *fp);
2430045 int      ferror     (FILE *fp);
2430046 int      fflush     (FILE *fp);
2430047 int      fgetc      (FILE *fp);
2430048 int      fgetpos    (FILE *restrict fp, fpos_t *restrict pos);
2430049 char    *fgets      (char *restrict string, int n, FILE *restrict fp);
2430050 int      fileno     (FILE *fp);
2430051 FILE    *fopen      (const char *path, const char *mode);
2430052 int      fprintf    (FILE *fp, char *restrict format, ...);
2430053 int      fputc      (int c, FILE *fp);
2430054 int      fputs      (const char *restrict string, FILE *restrict fp);
2430055 size_t   fread      (void *restrict buffer, size_t size, size_t nmemb,
2430056                      FILE *restrict fp);
2430057 FILE    *freopen    (const char *restrict path,
2430058                      const char *restrict mode,
2430059                      FILE *restrict fp);
2430060 int      fscanf     (FILE *restrict fp, const char *restrict format,
2430061                      ...);
2430062 int      fseek      (FILE *fp, long int offset, int whence);
2430063 int      fsetpos    (FILE *fp, const fpos_t *pos);
2430064 long int ftell      (FILE *fp);
2430065 off_t    ftello     (FILE *fp);
2430066 size_t   fwrite     (const void *restrict buffer, size_t size,
2430067                      size_t nmemb, FILE *restrict fp);
2430068 #define  getc(p)    (fgetc (p))
2430069 int      getchar    (void);
2430070 char    *gets       (char *string);
2430071 void     perror     (const char *string);
2430072 int      printf     (const char *restrict format, ...);
2430073 #define  putc(c, p) (fputc ((c), (p)))
2430074 #define  putchar(c) (fputc ((c), (stdout)))
2430075 int      puts       (const char *string);
2430076 void     rewind     (FILE *fp);
2430077 int      scanf      (const char *restrict format, ...);
2430078 void     setbuf     (FILE *restrict fp, char *restrict buffer);
2430079 int      setvbuf    (FILE *restrict fp, char *restrict buffer,
2430080                      int buf_mode, size_t size);
2430081 int      snprintf   (char *restrict string, size_t size,
2430082                      const char *restrict format, ...);
2430083 int      sprintf    (char *restrict string, const char *restrict format,
```

```
2430084                         ...);
2430085 int       sscanf      (char *restrict string, const char *restrict format,
2430086                         ...);
2430087 int       vfprintf    (FILE *fp, char *restrict format, va_list arg);
2430088 int       vfscanf     (FILE *restrict fp, const char *restrict format,
2430089                         va_list arg);
2430090 int       vprintf     (const char *restrict format, va_list arg);
2430091 int       vscanf      (const char *restrict format, va_list ap);
2430092 int       vsnprintf   (char *restrict string, size_t size,
2430093                         const char *restrict format, va_list arg);
2430094 int       vsprintf    (char *restrict string, const char *restrict format,
2430095                         va_list arg);
2430096 int       vsscanf     (const char *string, const char *format,
2430097                         va_list ap);
2430098
2430099 #endif
```

lib/stdio/FILE.c

«

Si veda la sezione u0.2.

```
2440001 #include <stdio.h>
2440002 //
2440003 // There must be room for at least `FOPEN_MAX' elements.
2440004 //
2440005 FILE _stream[FOPEN_MAX];
2440006 //---------------------------------------------------------------------
2440007 void
2440008 _stdio_stream_setup (void)
2440009 {
2440010     _stream[0].fdn   = 0;
2440011     _stream[0].error = 0;
2440012     _stream[0].eof   = 0;
2440013
2440014     _stream[1].fdn   = 1;
2440015     _stream[1].error = 0;
2440016     _stream[1].eof   = 0;
2440017
2440018     _stream[2].fdn   = 2;
2440019     _stream[2].error = 0;
2440020     _stream[2].eof   = 0;
2440021 }
```

lib/stdio/clearerr.c

«

Si veda la sezione u0.9.

```
2450001  #include <stdio.h>
2450002  //-------------------------------------------------------------------
2450003  void
2450004  clearerr (FILE *fp)
2450005  {
2450006      if (fp != NULL)
2450007        {
2450008          fp->error = 0;
2450009          fp->eof   = 0;
2450010        }
2450011  }
```

lib/stdio/fclose.c

«

Si veda la sezione u0.27.

```
2460001  #include <stdio.h>
2460002  #include <unistd.h>
2460003  //-------------------------------------------------------------------
2460004  int
2460005  fclose (FILE *fp)
2460006  {
2460007      return (close (fp->fdn));
2460008  }
```

lib/stdio/feof.c

«

Si veda la sezione u0.28.

```
2470001  #include <stdio.h>
2470002  //-------------------------------------------------------------------
2470003  int
2470004  feof (FILE *fp)
2470005  {
2470006      if (fp != NULL)
2470007        {
2470008          return (fp->eof);
2470009        }
```

```
2470010     return (0);
2470011 }
```

lib/stdio/ferror.c

«

Si veda la sezione u0.29.

```
2480001 #include <stdio.h>
2480002 //-----------------------------------------------------------------------
2480003 int
2480004 ferror (FILE *fp)
2480005 {
2480006     if (fp != NULL)
2480007       {
2480008         return (fp->error);
2480009       }
2480010     return (0);
2480011 }
```

lib/stdio/fflush.c

«

Si veda la sezione u0.30.

```
2490001 #include <stdio.h>
2490002 //-----------------------------------------------------------------------
2490003 int
2490004 fflush (FILE *fp)
2490005 {
2490006     //
2490007     // The os16 library does not have any buffered data.
2490008     //
2490009     return (0);
2490010 }
```

lib/stdio/fgetc.c

«

Si veda la sezione u0.31.

```
2500001 #include <stdio.h>
2500002 #include <sys/types.h>
2500003 #include <unistd.h>
2500004 //---------------------------------------------------------------------
2500005 int
2500006 fgetc (FILE *fp)
2500007 {
2500008     ssize_t size_read;
2500009     int     c;          // Character read.
2500010     //
2500011     for (c = 0;;)
2500012       {
2500013         size_read = read (fp->fdn, &c, (size_t) 1);
2500014         //
2500015         if (size_read <= 0)
2500016           {
2500017             //
2500018             // It is the end of file (zero) otherwise there is a
2500019             // problem (a negative value): return 'EOF'.
2500020             //
2500021             return (EOF);
2500022           }
2500023         //
2500024         // Valid read: end of scan.
2500025         //
2500026         return (c);
2500027       }
2500028 }
```

lib/stdio/fgetpos.c

«

Si veda la sezione u0.32.

```
2510001 #include <stdio.h>
2510002 //---------------------------------------------------------------------
2510003 int
2510004 fgetpos (FILE *restrict fp, fpos_t *restrict pos)
2510005 {
2510006     long int position;
```

```
2510007     //
2510008     if (fp != NULL)
2510009       {
2510010         position = ftell (fp);
2510011         if (position >= 0)
2510012           {
2510013             *pos = position;
2510014             return (0);
2510015           }
2510016       }
2510017     return (-1);
2510018 }
```

lib/stdio/fgets.c

«

Si veda la sezione u0.33.

```
2520001 #include <stdio.h>
2520002 #include <sys/types.h>
2520003 #include <unistd.h>
2520004 #include <stddef.h>
2520005 //----------------------------------------------------------------------
2520006 char *
2520007 fgets (char *restrict string, int n, FILE *restrict fp)
2520008 {
2520009     ssize_t size_read;
2520010     int     b;          // Index inside the string buffer.
2520011     //
2520012     for (b = 0; b < (n-1); b++, string[b] = 0)
2520013       {
2520014         size_read = read (fp->fdn, &string[b], (size_t) 1);
2520015         //
2520016         if (size_read <= 0)
2520017           {
2520018             //
2520019             // It is the end of file (zero) otherwise there is a
2520020             // problem (a negative value).
2520021             //
2520022             string[b] = 0;
2520023             break;
2520024           }
2520025         //
```

```
2520026           if (string[b] == '\n')
2520027             {
2520028               b++;
2520029               string[b] = 0;
2520030               break;
2520031             }
2520032         }
2520033       //
2520034       // If `b` is zero, nothing was read and `NULL' is returned.
2520035       //
2520036       if (b == 0)
2520037         {
2520038           return (NULL);
2520039         }
2520040       else
2520041         {
2520042           return (string);
2520043         }
2520044   }
```

lib/stdio/fileno.c

«

Si veda la sezione u0.34.

```
2530001   #include <stdio.h>
2530002   #include <errno.h>
2530003   //------------------------------------------------------------------------
2530004   int
2530005   fileno (FILE *fp)
2530006   {
2530007       if (fp != NULL)
2530008         {
2530009           return (fp->fdn);
2530010         }
2530011       errset (EBADF);                     // Bad file descriptor.
2530012       return (-1);
2530013   }
```

«

Si veda la sezione u0.35.

```
2540001  #include <fcntl.h>
2540002  #include <stdarg.h>
2540003  #include <stddef.h>
2540004  #include <string.h>
2540005  #include <errno.h>
2540006  #include <sys/os16.h>
2540007  #include <const.h>
2540008  #include <limits.h>
2540009  #include <stdio.h>
2540010
2540011  //-------------------------------------------------------------------
2540012  FILE *
2540013  fopen (const char *path, const char *mode)
2540014  {
2540015      int fdn;
2540016      //
2540017      if       (strcmp (mode, "r")    ||
2540018                strcmp (mode, "rb"))
2540019        {
2540020          fdn = open (path, O_RDONLY);
2540021        }
2540022      else if (strcmp (mode, "r+")   ||
2540023                strcmp (mode, "r+b")  ||
2540024                strcmp (mode, "rb+"))
2540025        {
2540026          fdn = open (path, O_RDWR);
2540027        }
2540028      else if (strcmp (mode, "w")   ||
2540029                strcmp (mode, "wb"))
2540030        {
2540031          fdn = open (path, O_WRONLY|O_CREAT|O_TRUNC, 0666);
2540032        }
2540033      else if (strcmp (mode, "w+")   ||
2540034                strcmp (mode, "w+b")  ||
2540035                strcmp (mode, "wb+"))
2540036        {
2540037          fdn = open (path, O_RDWR|O_CREAT|O_TRUNC, 0666);
2540038        }
2540039      else if (strcmp (mode, "a")   ||
2540040                strcmp (mode, "ab"))
```

```
2540041         {
2540042           fdn = open (path, O_WRONLY|O_APPEND|O_CREAT|O_TRUNC, 0666);
2540043         }
2540044       else if (strcmp (mode, "a+")  ||
2540045                strcmp (mode, "a+b") ||
2540046                strcmp (mode, "ab+"))
2540047         {
2540048           fdn = open (path, O_RDWR|O_APPEND|O_CREAT|O_TRUNC, 0666);
2540049         }
2540050       else
2540051         {
2540052           errset (EINVAL);                // Invalid argument.
2540053           return (NULL);
2540054         }
2540055       //
2540056       // Check the file descriptor returned.
2540057       //
2540058       if (fdn < 0)
2540059         {
2540060           //
2540061           // The variable 'errno' is already set.
2540062           //
2540063           errset (errno);
2540064           return (NULL);
2540065         }
2540066       //
2540067       // A valid file descriptor is available: convert it into a file
2540068       // stream. Please note that the file descriptor number must be
2540069       // saved inside the corresponding '_stream[]' array, because the
2540070       // file pointer do not have knowledge of the relative position
2540071       // inside the array.
2540072       //
2540073       _stream[fdn].fdn = fdn;     // Saved the file descriptor number.
2540074       //
2540075       return (&_stream[fdn]);     // Returned the file stream pointer.
2540076   }
```

lib/stdio/fprintf.c

«

Si veda la sezione u0.78.

```
2550001 #include <stdio.h>
2550002
2550003 //---------------------------------------------------------------------
2550004 int
2550005 fprintf (FILE *fp, char *restrict format, ...)
2550006 {
2550007     va_list ap;
2550008     va_start (ap, format);
2550009     return (vfprintf (fp, format, ap));
2550010 }
```

lib/stdio/fputc.c

«

Si veda la sezione u0.37.

```
2560001 #include <stdio.h>
2560002 #include <sys/types.h>
2560003 #include <sys/os16.h>
2560004 #include <string.h>
2560005 #include <unistd.h>
2560006 //---------------------------------------------------------------------
2560007 int
2560008 fputc (int c, FILE *fp)
2560009 {
2560010     ssize_t size_written;
2560011     char    character = (char) c;
2560012     size_written = write (fp->fdn, &character, (size_t) 1);
2560013     if (size_written < 0)
2560014       {
2560015         fp->eof = 1;
2560016         return (EOF);
2560017       }
2560018     return (c);
2560019 }
```

lib/stdio/fputs.c

«

Si veda la sezione u0.38.

```
2570001 #include <stdio.h>
2570002 #include <string.h>
2570003 //---------------------------------------------------------------------
2570004 int
2570005 fputs (const char *restrict string, FILE *restrict fp)
2570006 {
2570007     int      i; // Index inside the string to be printed.
2570008     int status;
2570009 
2570010     for (i = 0; i < strlen (string); i++)
2570011       {
2570012         status = fputc (string[i], fp);
2570013         if (status == EOF)
2570014           {
2570015             fp->eof = 1;
2570016             return (EOF);
2570017           }
2570018       }
2570019     return (0);
2570020 }
```

lib/stdio/fread.c

«

Si veda la sezione u0.39.

```
2580001 #include <unistd.h>
2580002 #include <stdio.h>
2580003 //---------------------------------------------------------------------
2580004 size_t
2580005 fread (void *restrict buffer, size_t size, size_t nmemb,
2580006        FILE *restrict fp)
2580007 {
2580008     ssize_t size_read;
2580009     size_read = read (fp->fdn, buffer, (size_t) (size * nmemb));
2580010     if (size_read == 0)
2580011       {
2580012         fp->eof = 1;
2580013         return ((size_t) 0);
2580014       }
```

```
2580015       else if (size_read < 0)
2580016         {
2580017           fp->error = 1;
2580018           return ((size_t) 0);
2580019         }
2580020       else
2580021         {
2580022           return ((size_t) (size_read / size));
2580023         }
2580024   }
```

lib/stdio/freopen.c

«

Si veda la sezione u0.35.

```
2590001   #include <fcntl.h>
2590002   #include <stdarg.h>
2590003   #include <stddef.h>
2590004   #include <string.h>
2590005   #include <errno.h>
2590006   #include <sys/os16.h>
2590007   #include <const.h>
2590008   #include <limits.h>
2590009   #include <stdio.h>
2590010
2590011   //-------------------------------------------------------------------------
2590012   FILE *
2590013   freopen (const char *restrict path, const char *restrict mode,
2590014            FILE *restrict fp)
2590015   {
2590016       int   status;
2590017       FILE *fp_new;
2590018       //
2590019       if (fp == NULL)
2590020         {
2590021           return (NULL);
2590022         }
2590023       //
2590024       status = fclose (fp);
2590025       if (status != 0)
2590026         {
2590027           fp->error = 1;
```

```
2590028           return (NULL);
2590029         }
2590030       //
2590031       fp_new = fopen (path, mode);
2590032       //
2590033       if (fp_new == NULL)
2590034         {
2590035           return (NULL);
2590036         }
2590037       //
2590038       if (fp_new != fp)
2590039         {
2590040           fclose (fp_new);
2590041           return (NULL);
2590042         }
2590043       //
2590044       return (fp_new);
2590045   }
```

lib/stdio/fscanf.c

«

Si veda la sezione u0.90.

```
2600001 #include <stdio.h>
2600002 //---------------------------------------------------------------------
2600003 int
2600004 fscanf (FILE *restrict fp, const char *restrict format, ...)
2600005 {
2600006     va_list ap;
2600007     va_start (ap, format);
2600008     return vfscanf (fp, format, ap);
2600009 }
```

lib/stdio/fseek.c

«

Si veda la sezione u0.43.

```
2610001 #include <stdio.h>
2610002 #include <unistd.h>
2610003 //---------------------------------------------------------------------
2610004 int
```

```
2610005 fseek (FILE *fp, long int offset, int whence)
2610006 {
2610007     off_t off_new;
2610008     off_new = lseek (fp->fdn, (off_t) offset, whence);
2610009     if (off_new < 0)
2610010       {
2610011         fp->error = 1;
2610012         return (-1);
2610013       }
2610014     else
2610015       {
2610016         fp->eof = 0;
2610017         return (0);
2610018       }
2610019 }
```

lib/stdio/fseeko.c

«

Si veda la sezione u0.43.

```
2620001 #include <stdio.h>
2620002 #include <unistd.h>
2620003 //------------------------------------------------------------------------
2620004 int
2620005 fseeko (FILE *fp, off_t offset, int whence)
2620006 {
2620007     off_t off_new;
2620008     off_new = lseek (fp->fdn, offset, whence);
2620009     if (off_new < 0)
2620010       {
2620011         fp->error = 1;
2620012         return (-1);
2620013       }
2620014     else
2620015       {
2620016         return (0);
2620017       }
2620018 }
```

lib/stdio/fsetpos.c

«

Si veda la sezione u0.32.

```
2630001 #include <stdio.h>
2630002 //-----------------------------------------------------------------------
2630003 int
2630004 fsetpos (FILE *restrict fp, fpos_t *restrict pos)
2630005 {
2630006     long int position;
2630007     //
2630008     if (fp != NULL)
2630009       {
2630010         position = fseek (fp, (long int) *pos, SEEK_SET);
2630011         if (position >= 0)
2630012           {
2630013             *pos = position;
2630014             return (0);
2630015           }
2630016       }
2630017     return (-1);
2630018 }
```

lib/stdio/ftell.c

«

Si veda la sezione u0.46.

```
2640001 #include <stdio.h>
2640002 #include <unistd.h>
2640003 //-----------------------------------------------------------------------
2640004 long int
2640005 ftell (FILE *fp)
2640006 {
2640007     return ((long int) lseek (fp->fdn, (off_t) 0, SEEK_CUR));
2640008 }
```

lib/stdio/ftello.c

«

Si veda la sezione u0.46.

```
2650001 #include <stdio.h>
2650002 #include <unistd.h>
2650003 //------------------------------------------------------------------------
2650004 off_t
2650005 ftello (FILE *fp)
2650006 {
2650007     return (lseek (fp->fdn, (off_t) 0, SEEK_CUR));
2650008 }
```

lib/stdio/fwrite.c

«

Si veda la sezione u0.48.

```
2660001 #include <unistd.h>
2660002 #include <stdio.h>
2660003 //------------------------------------------------------------------------
2660004 size_t
2660005 fwrite (const void *restrict buffer, size_t size, size_t nmemb,
2660006         FILE *restrict fp)
2660007 {
2660008     ssize_t size_written;
2660009     size_written = write (fp->fdn, buffer, (size_t) (size * nmemb));
2660010     if (size_written < 0)
2660011       {
2660012         fp->error = 1;
2660013         return ((size_t) 0);
2660014       }
2660015     else
2660016       {
2660017         return ((size_t) (size_written / size));
2660018       }
2660019 }
```

«

Si veda la sezione u0.31.

```
2670001 #include <stdio.h>
2670002 #include <sys/types.h>
2670003 #include <unistd.h>
2670004 //------------------------------------------------------------------------
2670005 int
2670006 getchar (void)
2670007 {
2670008     ssize_t size_read;
2670009     int     c;          // Character read.
2670010     //
2670011     for (c = 0;;)
2670012       {
2670013         size_read = read (STDIN_FILENO, &c, (size_t) 1);
2670014         //
2670015         if (size_read <= 0)
2670016           {
2670017             //
2670018             // It is the end of file (zero) otherwise there is a
2670019             // problem (a negative value): return 'EOF'.
2670020             //
2670021             _stream[STDIN_FILENO].eof = 1;
2670022             return (EOF);
2670023           }
2670024         //
2670025         // Valid read.
2670026         //
2670027         if (size_read == 0)
2670028           {
2670029             //
2670030             // If no character is ready inside the keyboard buffer, just
2670031             // retry.
2670032             //
2670033             continue;
2670034           }
2670035         //
2670036         // End of scan.
2670037         //
2670038         return (c);
2670039       }
2670040 }
```

«

Si veda la sezione u0.33.

```
2680001 #include <stdio.h>
2680002 #include <sys/types.h>
2680003 #include <unistd.h>
2680004 #include <stddef.h>
2680005 //------------------------------------------------------------------
2680006 char *
2680007 gets (char *string)
2680008 {
2680009     ssize_t size_read;
2680010     int     b;          // Index inside the string buffer.
2680011     //
2680012     for (b = 0;; b++, string[b] = 0)
2680013       {
2680014         size_read = read (STDIN_FILENO, &string[b], (size_t) 1);
2680015         //
2680016         if (size_read <= 0)
2680017           {
2680018             //
2680019             // It is the end of file (zero) otherwise there is a
2680020             // problem (a negative value).
2680021             //
2680022             _stream[STDIN_FILENO].eof = 1;
2680023             string[b] = 0;
2680024             break;
2680025           }
2680026         //
2680027         if (string[b] == '\n')
2680028           {
2680029             b++;
2680030             string[b] = 0;
2680031             break;
2680032           }
2680033       }
2680034     //
2680035     // If `b` is zero, nothing was read and `NULL' is returned.
2680036     //
2680037     if (b == 0)
2680038       {
2680039         return (NULL);
2680040       }
```

```
2680041       else
2680042         {
2680043           return (string);
2680044         }
2680045   }
```

lib/stdio/perror.c

«

Si veda la sezione u0.77.

```
2690001   #include <stdio.h>
2690002   #include <errno.h>
2690003   #include <stddef.h>
2690004   #include <string.h>
2690005   //-----------------------------------------------------------------------
2690006   void
2690007   perror (const char *string)
2690008   {
2690009       //
2690010       // If errno is zero, there is nothing to show.
2690011       //
2690012       if (errno == 0)
2690013         {
2690014           return;
2690015         }
2690016       //
2690017       // Show the string if there is one.
2690018       //
2690019       if (string != NULL && strlen (string) > 0)
2690020         {
2690021           printf ("%s: ", string);
2690022         }
2690023       //
2690024       // Show the translated error.
2690025       //
2690026       if (errfn[0] != 0 && errln != 0)
2690027         {
2690028           printf ("[%s:%u:%i] %s\n",
2690029                   errfn, errln, errno, strerror (errno));
2690030         }
2690031       else
2690032         {
```

```
2690033           printf ("[%i] %s\n", errno, strerror (errno));
2690034         }
2690035   }
```

lib/stdio/printf.c

«

Si veda la sezione u0.78.

```
2700001   #include <stdio.h>
2700002   //---------------------------------------------------------------------
2700003   int
2700004   printf (char *restrict format, ...)
2700005   {
2700006       va_list ap;
2700007       va_start (ap, format);
2700008       return (vprintf (format, ap));
2700009   }
```

lib/stdio/puts.c

«

Si veda la sezione u0.38.

```
2710001   #include <stdio.h>
2710002   //---------------------------------------------------------------------
2710003   int
2710004   puts (const char *string)
2710005   {
2710006       int status;
2710007       status = printf ("%s\n", string);
2710008       if (status < 0)
2710009         {
2710010           return (EOF);
2710011         }
2710012       else
2710013         {
2710014           return (status);
2710015         }
2710016   }
```

lib/stdio/rewind.c

«

Si veda la sezione u0.88.

```
2720001 #include <stdio.h>
2720002 //---------------------------------------------------------------------
2720003 void
2720004 rewind (FILE *fp)
2720005 {
2720006     (void) fseek (fp, 0L, SEEK_SET);
2720007     fp->error = 0;
2720008 }
```

lib/stdio/scanf.c

«

Si veda la sezione u0.90.

```
2730001 #include <stdio.h>
2730002 //---------------------------------------------------------------------
2730003 int
2730004 scanf (const char *restrict format, ...)
2730005 {
2730006     va_list ap;
2730007     va_start (ap, format);
2730008     return vfscanf (stdin, format, ap);
2730009 }
```

lib/stdio/setbuf.c

«

Si veda la sezione u0.93.

```
2740001 #include <stdio.h>
2740002 //---------------------------------------------------------------------
2740003 void
2740004 setbuf (FILE *restrict fp, char *restrict buffer)
2740005 {
2740006     //
2740007     // The os16 library does not have any buffered data.
2740008     //
2740009     return;
2740010 }
```

lib/stdio/setvbuf.c

«

Si veda la sezione u0.93.

```
2750001 #include <stdio.h>
2750002 //------------------------------------------------------------------------
2750003 int
2750004 setvbuf (FILE *restrict fp, char *restrict buffer, int buf_mode,
2750005           size_t size)
2750006 {
2750007     //
2750008     // The os16 library does not have any buffered data.
2750009     //
2750010     return (0);
2750011 }
```

lib/stdio/snprintf.c

«

Si veda la sezione u0.78.

```
2760001 #include <stdio.h>
2760002 #include <stdarg.h>
2760003 //------------------------------------------------------------------------
2760004 int
2760005 snprintf (char *restrict string, size_t size,
2760006            const char *restrict format, ...)
2760007 {
2760008     va_list ap;
2760009     va_start (ap, format);
2760010     return vsnprintf (string, size, format, ap);
2760011 }
```

lib/stdio/sprintf.c

«

Si veda la sezione u0.78.

```
2770001 #include <stdio.h>
2770002 #include <stdarg.h>
2770003 //------------------------------------------------------------------------
2770004 int
2770005 sprintf (char *restrict string, const char *restrict format,
2770006            ...)
```

```
2770007 {
2770008     va_list ap;
2770009     va_start (ap, format);
2770010     return vsnprintf (string, (size_t) BUFSIZ, format, ap);
2770011 }
```

lib/stdio/sscanf.c

«

Si veda la sezione u0.90.

```
2780001 #include <stdio.h>
2780002 //------------------------------------------------------------------------
2780003 int
2780004 sscanf (char *restrict string, const char *restrict format, ...)
2780005 {
2780006     va_list ap;
2780007     va_start (ap, format);
2780008     return vsscanf (string, format, ap);
2780009 }
```

lib/stdio/vfprintf.c

«

Si veda la sezione u0.128.

```
2790001 #include <stdio.h>
2790002 #include <sys/types.h>
2790003 #include <sys/os16.h>
2790004 #include <string.h>
2790005 #include <unistd.h>
2790006 //------------------------------------------------------------------------
2790007 int
2790008 vfprintf (FILE *fp, char *restrict format, va_list arg)
2790009 {
2790010     ssize_t     size_written;
2790011     size_t      size;
2790012     size_t      size_total;
2790013     int         status;
2790014     char        string[BUFSIZ];
2790015     char       *buffer = string;
2790016     //
2790017     buffer[0] = 0;
```

```
2790018       status    = vsprintf (buffer, format, arg);
2790019       //
2790020       size = strlen (buffer);
2790021       if (size >= BUFSIZ)
2790022         {
2790023           size = BUFSIZ;
2790024         }
2790025       //
2790026       for (size_total = 0, size_written = 0;
2790027            size_total < size;
2790028            size_total += size_written, buffer += size_written)
2790029         {
2790030           size_written = write (fp->fdn, buffer, size - size_total);
2790031           if (size_written < 0)
2790032             {
2790033               return (size_total);
2790034             }
2790035         }
2790036       return (size);
2790037   }
```

lib/stdio/vfscanf.c

«

Si veda la sezione u0.129.

```
2800001   #include <stdio.h>
2800002
2800003   //-------------------------------------------------------------------
2800004   int vfsscanf (FILE *restrict fp, const char *string,
2800005                 const char *restrict format, va_list ap);
2800006   //-------------------------------------------------------------------
2800007   int
2800008   vfscanf (FILE *restrict fp, const char *restrict format, va_list ap)
2800009   {
2800010       return (vfsscanf (fp, NULL, format, ap));
2800011   }
2800012   //-------------------------------------------------------------------
```

«

Si veda la sezione u0.129.

```
2810001 #include <stdint.h>
2810002 #include <stdbool.h>
2810003 #include <stdlib.h>
2810004 #include <string.h>
2810005 #include <stdio.h>
2810006 #include <stdarg.h>
2810007 #include <ctype.h>
2810008 #include <errno.h>
2810009 #include <stddef.h>
2810010 //-------------------------------------------------------------------
2810011 //
2810012 // This function is not standard and is able to do the work of both
2810013 // `vfscanf()' and `vsscanf()'.
2810014 //
2810015 //-------------------------------------------------------------------
2810016 #define WIDTH_MAX       64
2810017 //-------------------------------------------------------------------
2810018 static intmax_t strtointmax (const char *restrict string,
2810019                              char **restrict endptr, int base,
2810020                              size_t max_width);
2810021 static int      ass_or_eof  (int consumed, int assigned);
2810022 //-------------------------------------------------------------------
2810023 int
2810024 vfsscanf (FILE *restrict fp, const char *string,
2810025           const char *restrict format, va_list ap)
2810026 {
2810027     int                 f               = 0;            // Format index.
2810028     char                buffer[BUFSIZ];
2810029     const char         *input           = string;       // Default.
2810030     const char         *start           = input;        // Default.
2810031     char               *next            = NULL;
2810032     int                 scanned         = 0;
2810033     //
2810034     bool                stream          = 0;
2810035     bool                flag_star       = 0;
2810036     bool                specifier       = 0;
2810037     bool                specifier_flags = 0;
2810038     bool                specifier_width = 0;
2810039     bool                specifier_type  = 0;
2810040     bool                inverted        = 0;
```

```
2810041       //
2810042       char                *ptr_char;
2810043       signed char         *ptr_schar;
2810044       unsigned char       *ptr_uchar;
2810045       short int           *ptr_sshort;
2810046       unsigned short int  *ptr_ushort;
2810047       int                 *ptr_sint;
2810048       unsigned int        *ptr_uint;
2810049       long int            *ptr_slong;
2810050       unsigned long int   *ptr_ulong;
2810051       intmax_t            *ptr_simax;
2810052       uintmax_t           *ptr_uimax;
2810053       size_t              *ptr_size;
2810054       ptrdiff_t           *ptr_ptrdiff;
2810055       void               **ptr_void;
2810056       //
2810057       size_t               width;
2810058       char                 width_string[WIDTH_MAX+1];
2810059       int                  w;              // Index inside width string.
2810060       int                  assigned        = 0;    // Assignment counter.
2810061       int                  consumed        = 0;    // Consumed counter.
2810062       //
2810063       intmax_t             value_i;
2810064       uintmax_t            value_u;
2810065       //
2810066       const char          *end_format;
2810067       const char          *end_input;
2810068       int                  count;          // Generic counter.
2810069       int                  index;          // Generic index.
2810070       bool                 ascii[128];
2810071       //
2810072       void                *pstatus;
2810073       //
2810074       // Initialize some data.
2810075       //
2810076       width_string[0] = '\0';
2810077       end_format      = format + (strlen (format));
2810078       //
2810079       // Check arguments and find where input comes.
2810080       //
2810081       if (fp == NULL && (string == NULL || string[0] == 0))
2810082         {
2810083           errset (EINVAL);                // Invalid argument.
```

```
2810084           return (EOF);
2810085         }
2810086       //
2810087       if (fp != NULL && string != NULL && string[0] != 0)
2810088         {
2810089           errset (EINVAL);                // Invalid argument.
2810090           return (EOF);
2810091         }
2810092       //
2810093       if (fp != NULL)
2810094         {
2810095           stream = 1;
2810096         }
2810097       //
2810098       //
2810099       //
2810100       for (;;)
2810101         {
2810102           if (stream)
2810103             {
2810104               pstatus = fgets (buffer, BUFSIZ, fp);
2810105               //
2810106               if (pstatus == NULL)
2810107                 {
2810108                   return (ass_or_eof (consumed, assigned));
2810109                 }
2810110               //
2810111               input = buffer;
2810112               start = input;
2810113               next  = NULL;
2810114             }
2810115           //
2810116           // Calculate end input.
2810117           //
2810118           end_input = input + (strlen (input));
2810119           //
2810120           // Scan format and input strings. Index 'f' is not reset.
2810121           //
2810122           while (&format[f] < end_format && input < end_input)
2810123             {
2810124               if (!specifier)
2810125                 {
2810126                   //--------------- The context is not inside a specifier.
```

```
2810127                   if (isspace (format[f]))
2810128                     {
2810129                       //---------------------------------------- Space.
2810130                       while (isspace (*input))
2810131                         {
2810132                           input++;
2810133                         }
2810134                       //
2810135                       // Verify that the input string is not finished.
2810136                       //
2810137                       if (input[0] == 0)
2810138                         {
2810139                           //
2810140                           // As the input string is finished, the format
2810141                           // string index is not advanced, because there
2810142                           // might be more spaces on the next line (if
2810143                           // there is a next line, of course).
2810144                           //
2810145                           continue;
2810146                         }
2810147                       else
2810148                         {
2810149                           f++;
2810150                           continue;
2810151                         }
2810152                     }
2810153                   if (format[f] != '%')
2810154                     {
2810155                       //----------------------------- Ordinary character.
2810156                       if (format[f] == *input)
2810157                         {
2810158                           input++;
2810159                           f++;
2810160                           continue;
2810161                         }
2810162                       else
2810163                         {
2810164                           return (ass_or_eof (consumed, assigned));
2810165                         }
2810166                     }
2810167                   if (format[f] == '%' && format[f+1] == '%')
2810168                     {
2810169                       //------------------------- Matching a literal '%'.
```

```
2810170                       f++;
2810171                       if (format[f] == *input)
2810172                         {
2810173                           input++;
2810174                           f++;
2810175                           continue;
2810176                         }
2810177                       else
2810178                         {
2810179                           return (ass_or_eof (consumed, assigned));
2810180                         }
2810181                     }
2810182                   if (format[f] == '%')
2810183                     {
2810184                       //------------------------- Percent of a specifier.
2810185                       f++;
2810186                       specifier       = 1;
2810187                       specifier_flags = 1;
2810188                       continue;
2810189                     }
2810190                 }
2810191               //
2810192               if (specifier && specifier_flags)
2810193                 {
2810194                   //--------------- The context is inside specifier flags.
2810195                   if (format[f] == '*')
2810196                     {
2810197                       //--------------------- Assignment suppression star.
2810198                       flag_star = 1;
2810199                       f++;
2810200                     }
2810201                   else
2810202                     {
2810203                       //----- End of flags and begin of specifier length.
2810204                       specifier_flags = 0;
2810205                       specifier_width = 1;
2810206                     }
2810207                 }
2810208               //
2810209               if (specifier && specifier_width)
2810210                 {
2810211                   //------------- The context is inside a specifier width.
2810212                   for (w = 0;
```

```
2810213                          format[f] >= '0'
2810214                       && format[f] <= '9'
2810215                       && w < WIDTH_MAX;
2810216                      w++)
2810217                    {
2810218                      width_string[w] = format[f];
2810219                      f++;
2810220                    }
2810221                  width_string[w] = '\0';
2810222                  width = atoi (width_string);
2810223                  if (width > WIDTH_MAX)
2810224                    {
2810225                      width = WIDTH_MAX;
2810226                    }
2810227                  //
2810228                  // A zero width means an unspecified limit for the field
2810229                  // length.
2810230                  //
2810231                  //--------- End of spec. width and begin of spec. type.
2810232                  specifier_width = 0;
2810233                  specifier_type  = 1;
2810234                }
2810235              //
2810236              if (specifier && specifier_type)
2810237                {
2810238                  //
2810239                  // Specifiers with length modifier.
2810240                  //
2810241                  if (format[f] == 'h' && format[f+1] == 'h')
2810242                    {
2810243                      //------------------------------------------ char.
2810244                      if      (format[f+2] == 'd')
2810245                        {
2810246                          //----------------------- signed char, base 10.
2810247                          value_i = strtointmax (input, &next, 10, width);
2810248                          if (input == next)
2810249                            {
2810250                              return (ass_or_eof (consumed, assigned));
2810251                            }
2810252                          consumed++;
2810253                          if (!flag_star)
2810254                            {
2810255                              ptr_schar = va_arg (ap, signed char *);
```

```
2810256                             *ptr_schar = value_i;
2810257                              assigned++;
2810258                             }
2810259                           f += 3;
2810260                           input = next;
2810261                         }
2810262                       else if (format[f+2] == 'i')
2810263                         {
2810264                           //------------------- signed char, base unknown.
2810265                           value_i = strtointmax (input, &next, 0, width);
2810266                           if (input == next)
2810267                             {
2810268                               return (ass_or_eof (consumed, assigned));
2810269                             }
2810270                           consumed++;
2810271                           if (!flag_star)
2810272                             {
2810273                               ptr_schar = va_arg (ap, signed char *);
2810274                              *ptr_schar = value_i;
2810275                               assigned++;
2810276                             }
2810277                           f += 3;
2810278                           input = next;
2810279                         }
2810280                       else if (format[f+2] == 'o')
2810281                         {
2810282                           //------------------------- signed char, base 8.
2810283                           value_i = strtointmax (input, &next, 8, width);
2810284                           if (input == next)
2810285                             {
2810286                               return (ass_or_eof (consumed, assigned));
2810287                             }
2810288                           consumed++;
2810289                           if (!flag_star)
2810290                             {
2810291                               ptr_schar = va_arg (ap, signed char *);
2810292                              *ptr_schar = value_i;
2810293                               assigned++;
2810294                             }
2810295                           f += 3;
2810296                           input = next;
2810297                         }
2810298                       else if (format[f+2] == 'u')
```

```
2810299                         {
2810300                           //---------------------- unsigned char, base 10.
2810301                           value_u = strtointmax (input, &next, 10, width);
2810302                           if (input == next)
2810303                             {
2810304                               return (ass_or_eof (consumed, assigned));
2810305                             }
2810306                           consumed++;
2810307                           if (!flag_star)
2810308                             {
2810309                               ptr_uchar = va_arg (ap, unsigned char *);
2810310                              *ptr_uchar = value_u;
2810311                               assigned++;
2810312                             }
2810313                           f += 3;
2810314                           input = next;
2810315                         }
2810316                       else if (format[f+2] == 'x' || format[f+2] == 'X')
2810317                         {
2810318                           //------------------------ signed char, base 16.
2810319                           value_i = strtointmax (input, &next, 16, width);
2810320                           if (input == next)
2810321                             {
2810322                               return (ass_or_eof (consumed, assigned));
2810323                             }
2810324                           consumed++;
2810325                           if (!flag_star)
2810326                             {
2810327                               ptr_schar = va_arg (ap, signed char *);
2810328                              *ptr_schar = value_i;
2810329                               assigned++;
2810330                             }
2810331                           f += 3;
2810332                           input = next;
2810333                         }
2810334                       else if (format[f+2] == 'n')
2810335                         {
2810336                           //----------- signed char, string index counter.
2810337                           ptr_schar = va_arg (ap, signed char *);
2810338                          *ptr_schar = (signed char)
2810339                                       (input - start + scanned);
2810340                           f += 3;
2810341                         }
```

```
                    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;
```

```
2810385                           input = next;
2810386                         }
2810387                       else if (format[f+1] == 'o')
2810388                         {
2810389                           //------------------------ signed short, base 8.
2810390                           value_i = strtointmax (input, &next, 8, width);
2810391                           if (input == next)
2810392                             {
2810393                               return (ass_or_eof (consumed, assigned));
2810394                             }
2810395                           consumed++;
2810396                           if (!flag_star)
2810397                             {
2810398                               ptr_sshort = va_arg (ap, signed short *);
2810399                              *ptr_sshort = value_i;
2810400                               assigned++;
2810401                             }
2810402                           f += 2;
2810403                           input = next;
2810404                         }
2810405                       else if (format[f+1] == 'u')
2810406                         {
2810407                           //---------------------- unsigned short, base 10.
2810408                           value_u = strtointmax (input, &next, 10, width);
2810409                           if (input == next)
2810410                             {
2810411                               return (ass_or_eof (consumed, assigned));
2810412                             }
2810413                           consumed++;
2810414                           if (!flag_star)
2810415                             {
2810416                               ptr_ushort = va_arg (ap, unsigned short *);
2810417                              *ptr_ushort = value_u;
2810418                               assigned++;
2810419                             }
2810420                           f += 2;
2810421                           input = next;
2810422                         }
2810423                       else if (format[f+1] == 'x' || format[f+2] == 'X')
2810424                         {
2810425                           //----------------------- signed short, base 16.
2810426                           value_i = strtointmax (input, &next, 16, width);
2810427                           if (input == next)
```

```
2810428                             {
2810429                               return (ass_or_eof (consumed, assigned));
2810430                             }
2810431                           consumed++;
2810432                           if (!flag_star)
2810433                             {
2810434                               ptr_sshort = va_arg (ap, signed short *);
2810435                              *ptr_sshort = value_i;
2810436                               assigned++;
2810437                             }
2810438                           f += 2;
2810439                           input = next;
2810440                         }
2810441                       else if (format[f+1] == 'n')
2810442                         {
2810443                           //----------- signed char, string index counter.
2810444                           ptr_sshort = va_arg (ap, signed short *);
2810445                          *ptr_sshort = (signed short)
2810446                                        (input - start + scanned);
2810447                           f += 2;
2810448                         }
2810449                       else
2810450                         {
2810451                           //------------ unsupported or unknown specifier.
2810452                           f += 1;
2810453                         }
2810454                     }
2810455                   //------------------------ There is no 'long long int'.
2810456                   else if (format[f] == 'l')
2810457                     {
2810458                       //---------------------------------------- long int.
2810459                       if      (format[f+1] == 'd')
2810460                         {
2810461                           //----------------------- signed long, base 10.
2810462                           value_i = strtointmax (input, &next, 10, width);
2810463                           if (input == next)
2810464                             {
2810465                               return (ass_or_eof (consumed, assigned));
2810466                             }
2810467                           consumed++;
2810468                           if (!flag_star)
2810469                             {
2810470                               ptr_slong = va_arg (ap, signed long *);
```

```
2810471                             *ptr_slong = value_i;
2810472                              assigned++;
2810473                            }
2810474                          f += 2;
2810475                          input = next;
2810476                        }
2810477                      else if (format[f+1] == 'i')
2810478                        {
2810479                          //------------------ signed long, base unknown.
2810480                          value_i = strtointmax (input, &next, 0, width);
2810481                          if (input == next)
2810482                            {
2810483                              return (ass_or_eof (consumed, assigned));
2810484                            }
2810485                          consumed++;
2810486                          if (!flag_star)
2810487                            {
2810488                              ptr_slong = va_arg (ap, signed long *);
2810489                             *ptr_slong = value_i;
2810490                              assigned++;
2810491                            }
2810492                          f += 2;
2810493                          input = next;
2810494                        }
2810495                      else if (format[f+1] == 'o')
2810496                        {
2810497                          //------------------------ signed long, base 8.
2810498                          value_i = strtointmax (input, &next, 8, width);
2810499                          if (input == next)
2810500                            {
2810501                              return (ass_or_eof (consumed, assigned));
2810502                            }
2810503                          consumed++;
2810504                          if (!flag_star)
2810505                            {
2810506                              ptr_slong = va_arg (ap, signed long *);
2810507                             *ptr_slong = value_i;
2810508                              assigned++;
2810509                            }
2810510                          f += 2;
2810511                          input = next;
2810512                        }
2810513                      else if (format[f+1] == 'u')
```

```
2810514                         {
2810515                           //---------------------- unsigned long, base 10.
2810516                           value_u = strtointmax (input, &next, 10, width);
2810517                           if (input == next)
2810518                             {
2810519                               return (ass_or_eof (consumed, assigned));
2810520                             }
2810521                           consumed++;
2810522                           if (!flag_star)
2810523                             {
2810524                               ptr_ulong = va_arg (ap, unsigned long *);
2810525                              *ptr_ulong = value_u;
2810526                               assigned++;
2810527                             }
2810528                           f += 2;
2810529                           input = next;
2810530                         }
2810531                       else if (format[f+1] == 'x' || format[f+2] == 'X')
2810532                         {
2810533                           //------------------------ signed long, base 16.
2810534                           value_i = strtointmax (input, &next, 16, width);
2810535                           if (input == next)
2810536                             {
2810537                               return (ass_or_eof (consumed, assigned));
2810538                             }
2810539                           consumed++;
2810540                           if (!flag_star)
2810541                             {
2810542                               ptr_slong = va_arg (ap, signed long *);
2810543                              *ptr_slong = value_i;
2810544                               assigned++;
2810545                             }
2810546                           f += 2;
2810547                           input = next;
2810548                         }
2810549                       else if (format[f+1] == 'n')
2810550                         {
2810551                           //----------- signed char, string index counter.
2810552                           ptr_slong = va_arg (ap, signed long *);
2810553                          *ptr_slong = (signed long)
2810554                                       (input - start + scanned);
2810555                           f += 2;
2810556                         }
```

```
2810557                         else
2810558                           {
2810559                             //------------ unsupported or unknown specifier.
2810560                             f += 1;
2810561                           }
2810562                       }
2810563                     else if (format[f] == 'j')
2810564                       {
2810565                         //--------------------------------------- intmax_t.
2810566                         if      (format[f+1] == 'd')
2810567                           {
2810568                             //--------------------------- intmax_t, base 10.
2810569                             value_i = strtointmax (input, &next, 10, width);
2810570                             if (input == next)
2810571                               {
2810572                                 return (ass_or_eof (consumed, assigned));
2810573                               }
2810574                             consumed++;
2810575                             if (!flag_star)
2810576                               {
2810577                                 ptr_simax = va_arg (ap, intmax_t *);
2810578                                *ptr_simax = value_i;
2810579                                 assigned++;
2810580                               }
2810581                             f += 2;
2810582                             input = next;
2810583                           }
2810584                         else if (format[f+1] == 'i')
2810585                           {
2810586                             //---------------------- intmax_t, base unknown.
2810587                             value_i = strtointmax (input, &next, 0, width);
2810588                             if (input == next)
2810589                               {
2810590                                 return (ass_or_eof (consumed, assigned));
2810591                               }
2810592                             consumed++;
2810593                             if (!flag_star)
2810594                               {
2810595                                 ptr_simax = va_arg (ap, intmax_t *);
2810596                                *ptr_simax = value_i;
2810597                                 assigned++;
2810598                               }
2810599                             f += 2;
```

```
2810600                         input = next;
2810601                       }
2810602                     else if (format[f+1] == 'o')
2810603                       {
2810604                         //--------------------------- intmax_t, base 8.
2810605                         value_i = strtointmax (input, &next, 8, width);
2810606                         if (input == next)
2810607                           {
2810608                             return (ass_or_eof (consumed, assigned));
2810609                           }
2810610                         consumed++;
2810611                         if (!flag_star)
2810612                           {
2810613                             ptr_simax = va_arg (ap, intmax_t *);
2810614                            *ptr_simax = value_i;
2810615                             assigned++;
2810616                           }
2810617                         f += 2;
2810618                         input = next;
2810619                       }
2810620                     else if (format[f+1] == 'u')
2810621                       {
2810622                         //-------------------------- uintmax_t, base 10.
2810623                         value_u = strtointmax (input, &next, 10, width);
2810624                         if (input == next)
2810625                           {
2810626                             return (ass_or_eof (consumed, assigned));
2810627                           }
2810628                         consumed++;
2810629                         if (!flag_star)
2810630                           {
2810631                             ptr_uimax = va_arg (ap, uintmax_t *);
2810632                            *ptr_uimax = value_u;
2810633                             assigned++;
2810634                           }
2810635                         f += 2;
2810636                         input = next;
2810637                       }
2810638                     else if (format[f+1] == 'x' || format[f+2] == 'X')
2810639                       {
2810640                         //--------------------------- intmax_t, base 16.
2810641                         value_i = strtointmax (input, &next, 16, width);
2810642                         if (input == next)
```

```
2810643                             {
2810644                               return (ass_or_eof (consumed, assigned));
2810645                             }
2810646                           consumed++;
2810647                           if (!flag_star)
2810648                             {
2810649                               ptr_simax = va_arg (ap, intmax_t *);
2810650                              *ptr_simax = value_i;
2810651                               assigned++;
2810652                             }
2810653                           f += 2;
2810654                           input = next;
2810655                         }
2810656                       else if (format[f+1] == 'n')
2810657                         {
2810658                           //----------- signed char, string index counter.
2810659                           ptr_simax = va_arg (ap, intmax_t *);
2810660                          *ptr_simax = (intmax_t)
2810661                                       (input - start + scanned);
2810662                           f += 2;
2810663                         }
2810664                       else
2810665                         {
2810666                           //------------ unsupported or unknown specifier.
2810667                           f += 1;
2810668                         }
2810669                     }
2810670                   else if (format[f] == 'z')
2810671                     {
2810672                       //---------------------------------------- size_t.
2810673                       if      (format[f+1] == 'd')
2810674                         {
2810675                           //----------------------------- size_t, base 10.
2810676                           value_i = strtointmax (input, &next, 10, width);
2810677                           if (input == next)
2810678                             {
2810679                               return (ass_or_eof (consumed, assigned));
2810680                             }
2810681                           consumed++;
2810682                           if (!flag_star)
2810683                             {
2810684                               ptr_size = va_arg (ap, size_t *);
2810685                              *ptr_size = value_i;
```

```
2810686                             assigned++;
2810687                           }
2810688                         f += 2;
2810689                         input = next;
2810690                       }
2810691                     else if (format[f+1] == 'i')
2810692                       {
2810693                         //------------------------ size_t, base unknown.
2810694                         value_i = strtointmax (input, &next, 0, width);
2810695                         if (input == next)
2810696                           {
2810697                             return (ass_or_eof (consumed, assigned));
2810698                           }
2810699                         consumed++;
2810700                         if (!flag_star)
2810701                           {
2810702                             ptr_size = va_arg (ap, size_t *);
2810703                            *ptr_size = value_i;
2810704                             assigned++;
2810705                           }
2810706                         f += 2;
2810707                         input = next;
2810708                       }
2810709                     else if (format[f+1] == 'o')
2810710                       {
2810711                         //----------------------------- size_t, base 8.
2810712                         value_i = strtointmax (input, &next, 8, width);
2810713                         if (input == next)
2810714                           {
2810715                             return (ass_or_eof (consumed, assigned));
2810716                           }
2810717                         consumed++;
2810718                         if (!flag_star)
2810719                           {
2810720                             ptr_size = va_arg (ap, size_t *);
2810721                            *ptr_size = value_i;
2810722                             assigned++;
2810723                           }
2810724                         f += 2;
2810725                         input = next;
2810726                       }
2810727                     else if (format[f+1] == 'u')
2810728                       {
```

```
2810729                         //---------------------------- size_t, base 10.
2810730                         value_u = strtointmax (input, &next, 10, width);
2810731                         if (input == next)
2810732                           {
2810733                             return (ass_or_eof (consumed, assigned));
2810734                           }
2810735                         consumed++;
2810736                         if (!flag_star)
2810737                           {
2810738                             ptr_size = va_arg (ap, size_t *);
2810739                            *ptr_size = value_u;
2810740                             assigned++;
2810741                           }
2810742                         f += 2;
2810743                         input = next;
2810744                       }
2810745                     else if (format[f+1] == 'x' || format[f+2] == 'X')
2810746                       {
2810747                         //---------------------------- size_t, base 16.
2810748                         value_i = strtointmax (input, &next, 16, width);
2810749                         if (input == next)
2810750                           {
2810751                             return (ass_or_eof (consumed, assigned));
2810752                           }
2810753                         consumed++;
2810754                         if (!flag_star)
2810755                           {
2810756                             ptr_size = va_arg (ap, size_t *);
2810757                            *ptr_size = value_i;
2810758                             assigned++;
2810759                           }
2810760                         f += 2;
2810761                         input = next;
2810762                       }
2810763                     else if (format[f+1] == 'n')
2810764                       {
2810765                         //----------- signed char, string index counter.
2810766                         ptr_size = va_arg (ap, size_t *);
2810767                        *ptr_size = (size_t) (input - start + scanned);
2810768                         f += 2;
2810769                       }
2810770                     else
2810771                       {
```

```
2810772                         //------------ unsupported or unknown specifier.
2810773                         f += 1;
2810774                       }
2810775                   }
2810776                 else if (format[f] == 't')
2810777                   {
2810778                     //-------------------------------------- ptrdiff_t.
2810779                     if      (format[f+1] == 'd')
2810780                       {
2810781                         //------------------------- ptrdiff_t, base 10.
2810782                         value_i = strtointmax (input, &next, 10, width);
2810783                         if (input == next)
2810784                           {
2810785                             return (ass_or_eof (consumed, assigned));
2810786                           }
2810787                         consumed++;
2810788                         if (!flag_star)
2810789                           {
2810790                             ptr_ptrdiff = va_arg (ap, ptrdiff_t *);
2810791                            *ptr_ptrdiff = value_i;
2810792                             assigned++;
2810793                           }
2810794                         f += 2;
2810795                         input = next;
2810796                       }
2810797                     else if (format[f+1] == 'i')
2810798                       {
2810799                         //--------------------- ptrdiff_t, base unknown.
2810800                         value_i = strtointmax (input, &next, 0, width);
2810801                         if (input == next)
2810802                           {
2810803                             return (ass_or_eof (consumed, assigned));
2810804                           }
2810805                         consumed++;
2810806                         if (!flag_star)
2810807                           {
2810808                             ptr_ptrdiff = va_arg (ap, ptrdiff_t *);
2810809                            *ptr_ptrdiff = value_i;
2810810                             assigned++;
2810811                           }
2810812                         f += 2;
2810813                         input = next;
2810814                       }
```

```
2810815                         else if (format[f+1] == 'o')
2810816                           {
2810817                             //--------------------------- ptrdiff_t, base 8.
2810818                             value_i = strtointmax (input, &next, 8, width);
2810819                             if (input == next)
2810820                               {
2810821                                 return (ass_or_eof (consumed, assigned));
2810822                               }
2810823                             consumed++;
2810824                             if (!flag_star)
2810825                               {
2810826                                 ptr_ptrdiff = va_arg (ap, ptrdiff_t *);
2810827                                *ptr_ptrdiff = value_i;
2810828                                 assigned++;
2810829                               }
2810830                             f += 2;
2810831                             input = next;
2810832                           }
2810833                         else if (format[f+1] == 'u')
2810834                           {
2810835                             //--------------------------- ptrdiff_t, base 10.
2810836                             value_u = strtointmax (input, &next, 10, width);
2810837                             if (input == next)
2810838                               {
2810839                                 return (ass_or_eof (consumed, assigned));
2810840                               }
2810841                             consumed++;
2810842                             if (!flag_star)
2810843                               {
2810844                                 ptr_ptrdiff = va_arg (ap, ptrdiff_t *);
2810845                                *ptr_ptrdiff = value_u;
2810846                                 assigned++;
2810847                               }
2810848                             f += 2;
2810849                             input = next;
2810850                           }
2810851                         else if (format[f+1] == 'x' || format[f+2] == 'X')
2810852                           {
2810853                             //--------------------------- ptrdiff_t, base 16.
2810854                             value_i = strtointmax (input, &next, 16, width);
2810855                             if (input == next)
2810856                               {
2810857                                 return (ass_or_eof (consumed, assigned));
```

```
2810858                             }
2810859                           consumed++;
2810860                           if (!flag_star)
2810861                             {
2810862                               ptr_ptrdiff = va_arg (ap, ptrdiff_t *);
2810863                              *ptr_ptrdiff = value_i;
2810864                               assigned++;
2810865                             }
2810866                           f += 2;
2810867                           input = next;
2810868                         }
2810869                       else if (format[f+1] == 'n')
2810870                         {
2810871                           //----------- signed char, string index counter.
2810872                           ptr_ptrdiff = va_arg (ap, ptrdiff_t *);
2810873                          *ptr_ptrdiff = (ptrdiff_t)
2810874                                         (input - start + scanned);
2810875                           f += 2;
2810876                         }
2810877                       else
2810878                         {
2810879                           //------------ unsupported or unknown specifier.
2810880                           f += 1;
2810881                         }
2810882                     }
2810883                   //
2810884                   // Specifiers with no length modifier.
2810885                   //
2810886                   if      (format[f] == 'd')
2810887                     {
2810888                       //---------------------- signed short, base 10.
2810889                       value_i = strtointmax (input, &next, 10, width);
2810890                       if (input == next)
2810891                         {
2810892                           return (ass_or_eof (consumed, assigned));
2810893                         }
2810894                       consumed++;
2810895                       if (!flag_star)
2810896                         {
2810897                           ptr_sshort = va_arg (ap, signed short *);
2810898                          *ptr_sshort = value_i;
2810899                           assigned++;
2810900                         }
```

```
2810901                         f += 1;
2810902                         input = next;
2810903                       }
2810904                     else if (format[f] == 'i')
2810905                       {
2810906                         //------------------------ signed int, base unknown.
2810907                         value_i = strtointmax (input, &next, 0, width);
2810908                         if (input == next)
2810909                           {
2810910                             return (ass_or_eof (consumed, assigned));
2810911                           }
2810912                         consumed++;
2810913                         if (!flag_star)
2810914                           {
2810915                             ptr_sint = va_arg (ap, signed int *);
2810916                            *ptr_sint = value_i;
2810917                             assigned++;
2810918                           }
2810919                         f += 1;
2810920                         input = next;
2810921                       }
2810922                     else if (format[f] == 'o')
2810923                       {
2810924                         //------------------------------ signed int, base 8.
2810925                         value_i = strtointmax (input, &next, 8, width);
2810926                         if (input == next)
2810927                           {
2810928                             return (ass_or_eof (consumed, assigned));
2810929                           }
2810930                         consumed++;
2810931                         if (!flag_star)
2810932                           {
2810933                             ptr_sint = va_arg (ap, signed int *);
2810934                            *ptr_sint = value_i;
2810935                             assigned++;
2810936                           }
2810937                         f += 1;
2810938                         input = next;
2810939                       }
2810940                     else if (format[f] == 'u')
2810941                       {
2810942                         //------------------------ unsigned short, base 10.
2810943                         value_u = strtointmax (input, &next, 10, width);
```

```
2810944                         if (input == next)
2810945                           {
2810946                             return (ass_or_eof (consumed, assigned));
2810947                           }
2810948                         consumed++;
2810949                         if (!flag_star)
2810950                           {
2810951                             ptr_uint = va_arg (ap, unsigned int *);
2810952                            *ptr_uint = value_u;
2810953                             assigned++;
2810954                           }
2810955                         f += 1;
2810956                         input = next;
2810957                       }
2810958                     else if (format[f] == 'x' || format[f] == 'X')
2810959                       {
2810960                         //--------------------------- signed short, base 16.
2810961                         value_i = strtointmax (input, &next, 16, width);
2810962                         if (input == next)
2810963                           {
2810964                             return (ass_or_eof (consumed, assigned));
2810965                           }
2810966                         consumed++;
2810967                         if (!flag_star)
2810968                           {
2810969                             ptr_sint = va_arg (ap, signed int *);
2810970                            *ptr_sint = value_i;
2810971                             assigned++;
2810972                           }
2810973                         f += 1;
2810974                         input = next;
2810975                       }
2810976                     else if (format[f] == 'c')
2810977                       {
2810978                         //---------------------------------------- char[].
2810979                         if (width == 0) width = 1;
2810980                         //
2810981                         if (!flag_star) ptr_char = va_arg (ap, char *);
2810982                         //
2810983                         for (count = 0;
2810984                              width > 0 && *input != 0;
2810985                              width--, ptr_char++, input++)
2810986                           {
```

```
                        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;
```

```
2811030                         }
2811031                       //
2811032                       // Reset ascii array.
2811033                       //
2811034                       for (index = 0; index < 128; index++)
2811035                         {
2811036                           ascii[index] = inverted;
2811037                         }
2811038                       //
2811039                       //
2811040                       //
2811041                       for (count = 0;&format[f] < end_format; count++)
2811042                         {
2811043                           if (format[f] == ']' && count > 0)
2811044                             {
2811045                               break;
2811046                             }
2811047                           //
2811048                           // Check for an interval.
2811049                           //
2811050                           if (format[f+1] == '-'
2811051                            && format[f+2] != ']'
2811052                            && format[f+2] != 0)
2811053                             {
2811054                               //
2811055                               // Interval.
2811056                               //
2811057                               for (index = format[f];
2811058                                    index <= format[f+2];
2811059                                    index++)
2811060                                 {
2811061                                   ascii[index] = !inverted;
2811062                                 }
2811063                               f += 3;
2811064                               continue;
2811065                             }
2811066                           //
2811067                           // Single character.
2811068                           //
2811069                           index = format[f];
2811070                           ascii[index] = !inverted;
2811071                           f++;
2811072                         }
```

```
                        //
                        // 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)
                          {
```

```
2811116                         return (ass_or_eof (consumed, assigned));
2811117                       }
2811118                     consumed++;
2811119                     if (!flag_star)
2811120                       {
2811121                         ptr_void = va_arg (ap, void **);
2811122                        *ptr_void = (void *) ((int) value_i);
2811123                         assigned++;
2811124                       }
2811125                     f += 1;
2811126                     input = next;
2811127                   }
2811128                 else if (format[f] == 'n')
2811129                   {
2811130                     //--------------- signed char, string index counter.
2811131                     ptr_sint = va_arg (ap, signed int *);
2811132                    *ptr_sint = (signed char) (input - start + scanned);
2811133                     f += 1;
2811134                   }
2811135                 else
2811136                   {
2811137                     //---------------- unsupported or unknown specifier.
2811138                     ;
2811139                   }
2811140
2811141                 //-------------------------------------------------------
2811142                 // End of specifier.
2811143                 //-------------------------------------------------------
2811144
2811145                 width_string[0]     = '\0';
2811146                 specifier           = 0;
2811147                 specifier_flags     = 0;
2811148                 specifier_width     = 0;
2811149                 specifier_type      = 0;
2811150                 flag_star           = 0;
2811151
2811152               }
2811153           }
2811154         //
2811155         // The format or the input string is terminated.
2811156         //
2811157         if  (&format[f] < end_format && stream)
2811158           {
```

```
2811159               //
2811160               // Only the input string is finished, and the input comes
2811161               // from a stream, so another read will be done.
2811162               //
2811163               scanned += (int) (input - start);
2811164               continue;
2811165             }
2811166           //
2811167           // The format string is terminated.
2811168           //
2811169           return (ass_or_eof (consumed, assigned));
2811170         }
2811171   }
2811172   //------------------------------------------------------------------------
2811173   static intmax_t
2811174   strtointmax (const char *restrict string, char **endptr,
2811175                int base, size_t max_width)
2811176   {
2811177       int      i;
2811178       int      d;                  // Digits counter.
2811179       int      sign = +1;
2811180       intmax_t number;
2811181       intmax_t previous;
2811182       int      digit;
2811183       //
2811184       bool     flag_prefix_oct = 0;
2811185       bool     flag_prefix_exa = 0;
2811186       bool     flag_prefix_dec = 0;
2811187       //
2811188       // If the 'max_width' value is zero, fix it to the maximum
2811189       // that it can represent.
2811190       //
2811191       if (max_width == 0)
2811192         {
2811193           max_width = SIZE_MAX;
2811194         }
2811195       //
2811196       // Eat initial spaces, but if there are spaces, there is an
2811197       // error inside the calling function!
2811198       //
2811199       for (i = 0; isspace (string[i]); i++)
2811200         {
2811201           fprintf (stderr, "libc error: file \"%s\", line %i\n",
```

```
2811202                     __FILE__, __LINE__);
2811203           ;
2811204         }
2811205       //
2811206       // Check sign. The `max_width' counts also the sign, if there is
2811207       // one.
2811208       //
2811209       if (string[i] == '+')
2811210         {
2811211           sign = +1;
2811212           i++;
2811213           max_width--;
2811214         }
2811215       else if (string[i] == '-')
2811216         {
2811217           sign = -1;
2811218           i++;
2811219           max_width--;
2811220         }
2811221       //
2811222       // Check for prefix.
2811223       //
2811224       if (string[i] == '0')
2811225         {
2811226           if (string[i+1] == 'x' || string[i+1] == 'X')
2811227             {
2811228               flag_prefix_exa = 1;
2811229             }
2811230           if (isdigit (string[i+1]))
2811231             {
2811232               flag_prefix_oct = 1;
2811233             }
2811234         }
2811235       //
2811236       if (string[i] > '0' && string[i] <= '9')
2811237         {
2811238           flag_prefix_dec = 1;
2811239         }
2811240       //
2811241       // Check compatibility with requested base.
2811242       //
2811243       if (flag_prefix_exa)
2811244         {
```

```
2811245           if (base == 0)
2811246             {
2811247               base = 16;
2811248             }
2811249           else if (base == 16)
2811250             {
2811251               ;   // Ok.
2811252             }
2811253           else
2811254             {
2811255               //
2811256               // Incompatible sequence: only the initial zero is reported.
2811257               //
2811258               *endptr = &string[i+1];
2811259               return ((intmax_t) 0);
2811260             }
2811261           //
2811262           // Move on, after the '0x' prefix.
2811263           //
2811264           i += 2;
2811265         }
2811266       //
2811267       if (flag_prefix_oct)
2811268         {
2811269           if (base == 0)
2811270             {
2811271               base = 8;
2811272             }
2811273           //
2811274           // Move on, after the '0' prefix.
2811275           //
2811276           i += 1;
2811277         }
2811278       //
2811279       if (flag_prefix_dec)
2811280         {
2811281           if (base == 0)
2811282             {
2811283               base = 10;
2811284             }
2811285         }
2811286       //
2811287       // Scan the string.
```

```
2811288       //
2811289       for (d = 0, number = 0; d < max_width && string[i] != 0; i++, d++)
2811290         {
2811291           if      (string[i] >= '0' && string[i] <= '9')
2811292             {
2811293               digit = string[i] - '0';
2811294             }
2811295           else if (string[i] >= 'A' && string[i] <= 'F')
2811296             {
2811297               digit = string[i] - 'A' + 10;
2811298             }
2811299           else if (string[i] >= 'a' && string[i] <= 'f')
2811300             {
2811301               digit = string[i] - 'a' + 10;
2811302             }
2811303           else
2811304             {
2811305               digit = 999;
2811306             }
2811307           //
2811308           // Give a sign to the digit.
2811309           //
2811310           digit *= sign;
2811311           //
2811312           // Compare with the base.
2811313           //
2811314           if (base > (digit * sign))
2811315             {
2811316               //
2811317               // Check if the current digit can be safely computed.
2811318               //
2811319               previous = number;
2811320               number  *= base;
2811321               number  += digit;
2811322               if (number / base != previous)
2811323                 {
2811324                   //
2811325                   // Out of range.
2811326                   //
2811327                   *endptr = &string[i+1];
2811328                   errset (ERANGE);        // Result too large.
2811329                   if (sign > 0)
2811330                     {
```

```
2811331                       return (INTMAX_MAX);
2811332                     }
2811333                   else
2811334                     {
2811335                       return (INTMAX_MIN);
2811336                     }
2811337                 }
2811338             }
2811339           else
2811340             {
2811341               *endptr = &string[i];
2811342               return (number);
2811343             }
2811344         }
2811345       //
2811346       // The string is finished or the max digits length is reached.
2811347       //
2811348       *endptr = &string[i];
2811349       //
2811350       return (number);
2811351   }
2811352   //------------------------------------------------------------------------
2811353   static int
2811354   ass_or_eof (int consumed, int assigned)
2811355   {
2811356       if (consumed == 0)
2811357         {
2811358           return (EOF);
2811359         }
2811360       else
2811361         {
2811362           return (assigned);
2811363         }
2811364   }
2811365   //------------------------------------------------------------------------
```

lib/stdio/vprintf.c

«

Si veda la sezione u0.128.

```
2820001   #include <stdio.h>
2820002   #include <sys/types.h>
```

```
2820003 #include <sys/os16.h>
2820004 #include <string.h>
2820005 #include <unistd.h>
2820006 //------------------------------------------------------------------------
2820007 int
2820008 vprintf (char *restrict format, va_list arg)
2820009 {
2820010     ssize_t     size_written;
2820011     size_t      size;
2820012     size_t      size_total;
2820013     int         status;
2820014     char        string[BUFSIZ];
2820015     char       *buffer = string;
2820016
2820017     buffer[0] = 0;
2820018     status = vsprintf (buffer, format, arg);
2820019
2820020     size = strlen (buffer);
2820021     if (size >= BUFSIZ)
2820022       {
2820023         size = BUFSIZ;
2820024       }
2820025
2820026     for (size_total = 0, size_written = 0;
2820027          size_total < size;
2820028          size_total += size_written, buffer += size_written)
2820029       {
2820030         //
2820031         // Write to the standard output: file descriptor n. 1.
2820032         //
2820033         size_written = write (STDOUT_FILENO, buffer, size - size_total);
2820034         if (size_written < 0)
2820035           {
2820036             return (size_total);
2820037           }
2820038       }
2820039     return (size);
2820040 }
```

lib/stdio/vscanf.c

«

Si veda la sezione u0.129.

```
2830001 #include <stdio.h>
2830002 //------------------------------------------------------------------------
2830003 int
2830004 vscanf (const char *restrict format, va_list ap)
2830005 {
2830006     return (vfscanf (stdin, format, ap));
2830007 }
2830008 //------------------------------------------------------------------------
```

lib/stdio/vsnprintf.c

«

Si veda la sezione u0.128.

```
2840001 #include <stdint.h>
2840002 #include <stdbool.h>
2840003 #include <stdlib.h>
2840004 #include <string.h>
2840005 #include <stdio.h>
2840006 //------------------------------------------------------------------------
2840007 static size_t uimaxtoa      (uintmax_t integer, char *buffer, int base,
2840008                              int uppercase, size_t size);
2840009 static size_t imaxtoa       (intmax_t  integer, char *buffer, int base,
2840010                              int uppercase, size_t size);
2840011 static size_t simaxtoa      (intmax_t  integer, char *buffer, int base,
2840012                              int uppercase, size_t size);
2840013 static size_t uimaxtoa_fill (uintmax_t integer, char *buffer, int base,
2840014                              int uppercase, int width, int filler,
2840015                              int max);
2840016 static size_t imaxtoa_fill  (intmax_t  integer, char *buffer, int base,
2840017                              int uppercase, int width, int filler,
2840018                              int max);
2840019 static size_t simaxtoa_fill (intmax_t  integer, char *buffer, int base,
2840020                              int uppercase, int width, int filler,
2840021                              int max);
2840022 static size_t strtostr_fill (char *string, char *buffer, int width,
2840023                              int filler, int max);
2840024 //------------------------------------------------------------------------
2840025 int
2840026 vsnprintf (char *restrict string, size_t size,
```

```
2840027             const char *restrict format, va_list ap)
2840028 {
2840029     //
2840030     // We produce at most 'size-1' characters, + '\0'.
2840031     // 'size' is used also as the max size for internal
2840032     // strings, but only if it is not too big.
2840033     //
2840034     int        f                   = 0;
2840035     int        s                   = 0;
2840036     int        remain              = size - 1;
2840037     //
2840038     bool       specifier           = 0;
2840039     bool       specifier_flags     = 0;
2840040     bool       specifier_width     = 0;
2840041     bool       specifier_precision = 0;
2840042     bool       specifier_type      = 0;
2840043     //
2840044     bool       flag_plus           = 0;
2840045     bool       flag_minus          = 0;
2840046     bool       flag_space          = 0;
2840047     bool       flag_alternate      = 0;
2840048     bool       flag_zero           = 0;
2840049     //
2840050     int        alignment;
2840051     int        filler;
2840052     //
2840053     intmax_t   value_i;
2840054     uintmax_t  value_ui;
2840055     char      *value_cp;
2840056     //
2840057     size_t     width;
2840058     size_t     precision;
2840059     #define    str_size  BUFSIZ/2
2840060     char       width_string[str_size];
2840061     char       precision_string[str_size];
2840062     int        w;
2840063     int        p;
2840064     //
2840065     width_string[0]     = '\0';
2840066     precision_string[0] = '\0';
2840067     //
2840068     while (format[f] != 0 && s < (size - 1))
2840069       {
```

```
2840070           if (!specifier)
2840071             {
2840072               //------------------- The context is not inside a specifier.
2840073               if (format[f] != '%')
2840074                 {
2840075                   string[s] = format[f];
2840076                   s++;
2840077                   remain--;
2840078                   f++;
2840079                   continue;
2840080                 }
2840081               if (format[f] == '%' && format[f+1] == '%')
2840082                 {
2840083                   string[s] = '%';
2840084                   f++;
2840085                   f++;
2840086                   s++;
2840087                   remain--;
2840088                   continue;
2840089                 }
2840090               if (format[f] == '%')
2840091                 {
2840092                   f++;
2840093                   specifier = 1;
2840094                   specifier_flags = 1;
2840095                   continue;
2840096                 }
2840097             }
2840098           //
2840099           if (specifier && specifier_flags)
2840100             {
2840101               //------------------- The context is inside specifier flags.
2840102               if (format[f] == '+')
2840103                 {
2840104                   flag_plus = 1;
2840105                   f++;
2840106                   continue;
2840107                 }
2840108               else if (format[f] == '-')
2840109                 {
2840110                   flag_minus = 1;
2840111                   f++;
2840112                   continue;
```

```
2840113                 }
2840114               else if (format[f] == ' ')
2840115                 {
2840116                   flag_space = 1;
2840117                   f++;
2840118                   continue;
2840119                 }
2840120               else if (format[f] == '#')
2840121                 {
2840122                   flag_alternate = 1;
2840123                   f++;
2840124                   continue;
2840125                 }
2840126               else if (format[f] == '0')
2840127                 {
2840128                   flag_zero = 1;
2840129                   f++;
2840130                   continue;
2840131                 }
2840132               else
2840133                 {
2840134                   specifier_flags = 0;
2840135                   specifier_width = 1;
2840136                 }
2840137             }
2840138           //
2840139           if (specifier && specifier_width)
2840140             {
2840141               //------------------- The context is inside specifier width.
2840142               for (w = 0; format[f] >= '0' && format[f] <= '9'
2840143                    && w < str_size; w++)
2840144                 {
2840145                   width_string[w] = format[f];
2840146                   f++;
2840147                 }
2840148               width_string[w] = '\0';
2840149
2840150               specifier_width = 0;
2840151
2840152               if (format[f] == '.')
2840153                 {
2840154                   specifier_precision = 1;
2840155                   f++;
```

```
2840156                 }
2840157               else
2840158                 {
2840159                   specifier_precision = 0;
2840160                   specifier_type      = 1;
2840161                 }
2840162             }
2840163           //
2840164           if (specifier && specifier_precision)
2840165             {
2840166               //--------------- The context is inside specifier precision.
2840167               for (p = 0; format[f] >= '0' && format[f] <= '9'
2840168                    && p < str_size; p++)
2840169                 {
2840170                   precision_string[p] = format[f];
2840171                   p++;
2840172                 }
2840173               precision_string[p] = '\0';
2840174
2840175               specifier_precision = 0;
2840176               specifier_type      = 1;
2840177             }
2840178           //
2840179           if (specifier && specifier_type)
2840180             {
2840181               //-------------------- The context is inside specifier type.
2840182               width     = atoi (width_string);
2840183               precision = atoi (precision_string);
2840184                                 filler = ' ';
2840185               if (flag_zero)  filler = '0';
2840186               if (flag_space) filler = ' ';
2840187                                 alignment = width;
2840188               if (flag_minus)
2840189                 {
2840190                   alignment = -alignment;
2840191                   filler = ' ';   // The filler character cannot
2840192                                   // be zero, so it is black.
2840193                 }
2840194               //
2840195               if (format[f] == 'h' && format[f+1] == 'h')
2840196                 {
2840197                   if (format[f+2] == 'd' || format[f+2] == 'i')
2840198                     {
```

```
2840199                         //---------------------------- signed char, base 10.
2840200                         value_i = va_arg (ap, int);
2840201                         if (flag_plus)
2840202                           {
2840203                             s += simaxtoa_fill (value_i, &string[s], 10, 0,
2840204                                                 alignment, filler, remain);
2840205                           }
2840206                         else
2840207                           {
2840208                             s += imaxtoa_fill (value_i, &string[s], 10, 0,
2840209                                                alignment, filler, remain);
2840210                           }
2840211                         f += 3;
2840212                       }
2840213                     else if (format[f+2] == 'u')
2840214                       {
2840215                         //-------------------------- unsigned char, base 10.
2840216                         value_ui = va_arg (ap, unsigned int);
2840217                         s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
2840218                                             alignment, filler, remain);
2840219                         f += 3;
2840220                       }
2840221                     else if (format[f+2] == 'o')
2840222                       {
2840223                         //---------------------------- unsigned char, base 8.
2840224                         value_ui = va_arg (ap, unsigned int);
2840225                         s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840226                                             alignment, filler, remain);
2840227                         f += 3;
2840228                       }
2840229                     else if (format[f+2] == 'x')
2840230                       {
2840231                         //-------------------------- unsigned char, base 16.
2840232                         value_ui = va_arg (ap, unsigned int);
2840233                         s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840234                                             alignment, filler, remain);
2840235                         f += 3;
2840236                       }
2840237                     else if (format[f+2] == 'X')
2840238                       {
2840239                         //-------------------------- unsigned char, base 16.
2840240                         value_ui = va_arg (ap, unsigned int);
2840241                         s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
```

```
2840242                                       alignment, filler, remain);
2840243                     f += 3;
2840244                   }
2840245                 else
2840246                   {
2840247                     //---------------- unsupported or unknown specifier.
2840248                     f += 2;
2840249                   }
2840250               }
2840251             else if (format[f] == 'h')
2840252               {
2840253                 if (format[f+1] == 'd' || format[f+1] == 'i')
2840254                   {
2840255                     //------------------------------ short int, base 10.
2840256                     value_i = va_arg (ap, int);
2840257                     if (flag_plus)
2840258                       {
2840259                         s += simaxtoa_fill (value_i, &string[s], 10, 0,
2840260                                             alignment, filler, remain);
2840261                       }
2840262                     else
2840263                       {
2840264                         s += imaxtoa_fill (value_i, &string[s], 10, 0,
2840265                                            alignment, filler, remain);
2840266                       }
2840267                     f += 2;
2840268                   }
2840269                 else if (format[f+1] == 'u')
2840270                   {
2840271                     //--------------------- unsigned short int, base 10.
2840272                     value_ui = va_arg (ap, unsigned int);
2840273                     s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
2840274                                         alignment, filler, remain);
2840275                     f += 2;
2840276                   }
2840277                 else if (format[f+1] == 'o')
2840278                   {
2840279                     //---------------------- unsigned short int, base 8.
2840280                     value_ui = va_arg (ap, unsigned int);
2840281                     s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840282                                         alignment, filler, remain);
2840283                     f += 2;
2840284                   }
```

```
2840285                 else if (format[f+1] == 'x')
2840286                   {
2840287                     //--------------------- unsigned short int, base 16.
2840288                     value_ui = va_arg (ap, unsigned int);
2840289                     s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840290                                         alignment, filler, remain);
2840291                     f += 2;
2840292                   }
2840293                 else if (format[f+1] == 'X')
2840294                   {
2840295                     //--------------------- unsigned short int, base 16.
2840296                     value_ui = va_arg (ap, unsigned int);
2840297                     s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
2840298                                         alignment, filler, remain);
2840299                     f += 2;
2840300                   }
2840301                 else
2840302                   {
2840303                     //---------------- unsupported or unknown specifier.
2840304                     f += 1;
2840305                   }
2840306               }
2840307
2840308                 //-------------------------------------------------------
2840309                 // There is no `long long int'.
2840310                 //-------------------------------------------------------
2840311
2840312             else if (format[f] == 'l')
2840313               {
2840314                 if (format[f+1] == 'd' || format[f+1] == 'i')
2840315                   {
2840316                     //------------------------------- long int base 10.
2840317                     value_i = va_arg (ap, long int);
2840318                     if (flag_plus)
2840319                       {
2840320                         s += simaxtoa_fill (value_i, &string[s], 10, 0,
2840321                                             alignment, filler, remain);
2840322                       }
2840323                     else
2840324                       {
2840325                         s += imaxtoa_fill (value_i, &string[s], 10, 0,
2840326                                            alignment, filler, remain);
2840327                       }
```

```
2840328                     f += 2;
2840329                   }
2840330                 else if (format[f+1] == 'u')
2840331                   {
2840332                     //----------------------- Unsigned long int base 10.
2840333                     value_ui = va_arg (ap, unsigned long int);
2840334                     s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
2840335                                         alignment, filler, remain);
2840336                     f += 2;
2840337                   }
2840338                 else if (format[f+1] == 'o')
2840339                   {
2840340                     //----------------------- Unsigned long int base 8.
2840341                     value_ui = va_arg (ap, unsigned long int);
2840342                     s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840343                                         alignment, filler, remain);
2840344                     f += 2;
2840345                   }
2840346                 else if (format[f+1] == 'x')
2840347                   {
2840348                     //----------------------- Unsigned long int base 16.
2840349                     value_ui = va_arg (ap, unsigned long int);
2840350                     s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840351                                         alignment, filler, remain);
2840352                     f += 2;
2840353                   }
2840354                 else if (format[f+1] == 'X')
2840355                   {
2840356                     //----------------------- Unsigned long int base 16.
2840357                     value_ui = va_arg (ap, unsigned long int);
2840358                     s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
2840359                                         alignment, filler, remain);
2840360                     f += 2;
2840361                   }
2840362                 else
2840363                   {
2840364                     //---------------- unsupported or unknown specifier.
2840365                     f += 1;
2840366                   }
2840367               }
2840368             else if (format[f] == 'j')
2840369               {
2840370                 if (format[f+1] == 'd' || format[f+1] == 'i')
```

```
2840371                     {
2840372                       //------------------------------ intmax_t base 10.
2840373                       value_i = va_arg (ap, intmax_t);
2840374                       if (flag_plus)
2840375                         {
2840376                           s += simaxtoa_fill (value_i, &string[s], 10, 0,
2840377                                               alignment, filler, remain);
2840378                         }
2840379                       else
2840380                         {
2840381                           s += imaxtoa_fill (value_i, &string[s], 10, 0,
2840382                                              alignment, filler, remain);
2840383                         }
2840384                       f += 2;
2840385                     }
2840386                   else if (format[f+1] == 'u')
2840387                     {
2840388                       //------------------------------ uintmax_t base 10.
2840389                       value_ui = va_arg (ap, uintmax_t);
2840390                       s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
2840391                                           alignment, filler, remain);
2840392                       f += 2;
2840393                     }
2840394                   else if (format[f+1] == 'o')
2840395                     {
2840396                       //------------------------------ uintmax_t base 8.
2840397                       value_ui = va_arg (ap, uintmax_t);
2840398                       s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840399                                           alignment, filler, remain);
2840400                       f += 2;
2840401                     }
2840402                   else if (format[f+1] == 'x')
2840403                     {
2840404                       //------------------------------ uintmax_t base 16.
2840405                       value_ui = va_arg (ap, uintmax_t);
2840406                       s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840407                                           alignment, filler, remain);
2840408                       f += 2;
2840409                     }
2840410                   else if (format[f+1] == 'X')
2840411                     {
2840412                       //------------------------------ uintmax_t base 16.
2840413                       value_ui = va_arg (ap, uintmax_t);
```

```
2840414                       s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
2840415                                           alignment, filler, remain);
2840416                       f += 2;
2840417                     }
2840418                   else
2840419                     {
2840420                       //---------------- unsupported or unknown specifier.
2840421                       f += 1;
2840422                     }
2840423                 }
2840424               else if (format[f] == 'z')
2840425                 {
2840426                   if (format[f+1] == 'd'
2840427                       || format[f+1] == 'i'
2840428                       || format[f+1] == 'i')
2840429                     {
2840430                       //---------------------------------- size_t base 10.
2840431                       value_ui = va_arg (ap, unsigned long int);
2840432                       s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
2840433                                           alignment, filler, remain);
2840434                       f += 2;
2840435                     }
2840436                   else if (format[f+1] == 'o')
2840437                     {
2840438                       //---------------------------------- size_t base 8.
2840439                       value_ui = va_arg (ap, unsigned long int);
2840440                       s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840441                                           alignment, filler, remain);
2840442                       f += 2;
2840443                     }
2840444                   else if (format[f+1] == 'x')
2840445                     {
2840446                       //---------------------------------- size_t base 16.
2840447                       value_ui = va_arg (ap, unsigned long int);
2840448                       s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840449                                           alignment, filler, remain);
2840450                       f += 2;
2840451                     }
2840452                   else if (format[f+1] == 'X')
2840453                     {
2840454                       //---------------------------------- size_t base 16.
2840455                       value_ui = va_arg (ap, unsigned long int);
2840456                       s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
```

```
2840457                                       alignment, filler, remain);
2840458                     f += 2;
2840459                   }
2840460                 else
2840461                   {
2840462                     //---------------- unsupported or unknown specifier.
2840463                     f += 1;
2840464                   }
2840465               }
2840466             else if (format[f] == 't')
2840467               {
2840468                 if (format[f+1] == 'd' || format[f+1] == 'i')
2840469                   {
2840470                     //------------------------------ ptrdiff_t base 10.
2840471                     value_i = va_arg (ap, long int);
2840472                     if (flag_plus)
2840473                       {
2840474                         s += simaxtoa_fill (value_i, &string[s], 10, 0,
2840475                                             alignment, filler, remain);
2840476                       }
2840477                     else
2840478                       {
2840479                         s += imaxtoa_fill (value_i, &string[s], 10, 0,
2840480                                            alignment, filler, remain);
2840481                       }
2840482                     f += 2;
2840483                   }
2840484                 else if (format[f+1] == 'u')
2840485                   {
2840486                     //----------------- ptrdiff_t base 10, without sign.
2840487                     value_ui = va_arg (ap, unsigned long int);
2840488                     s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
2840489                                         alignment, filler, remain);
2840490                     f += 2;
2840491                   }
2840492                 else if (format[f+1] == 'o')
2840493                   {
2840494                     //------------------ ptrdiff_t base 8, without sign.
2840495                     value_ui = va_arg (ap, unsigned long int);
2840496                     s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840497                                         alignment, filler, remain);
2840498                     f += 2;
2840499                   }
```

```
2840500                   else if (format[f+1] == 'x')
2840501                     {
2840502                       //----------------- ptrdiff_t base 16, without sign.
2840503                       value_ui = va_arg (ap, unsigned long int);
2840504                       s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840505                                           alignment, filler, remain);
2840506                       f += 2;
2840507                     }
2840508                   else if (format[f+1] == 'X')
2840509                     {
2840510                       //----------------- ptrdiff_t base 16, without sign.
2840511                       value_ui = va_arg (ap, unsigned long int);
2840512                       s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
2840513                                           alignment, filler, remain);
2840514                       f += 2;
2840515                     }
2840516                   else
2840517                     {
2840518                       //---------------- unsupported or unknown specifier.
2840519                       f += 1;
2840520                     }
2840521                 }
2840522               if (format[f] == 'd' || format[f] == 'i')
2840523                 {
2840524                   //---------------------------------------- int base 10.
2840525                   value_i = va_arg (ap, int);
2840526                   if (flag_plus)
2840527                     {
2840528                       s += simaxtoa_fill (value_i, &string[s], 10, 0,
2840529                                           alignment, filler, remain);
2840530                     }
2840531                   else
2840532                     {
2840533                       s += imaxtoa_fill (value_i, &string[s], 10, 0,
2840534                                          alignment, filler, remain);
2840535                     }
2840536                   f += 1;
2840537                 }
2840538               else if (format[f] == 'u')
2840539                 {
2840540                   //------------------------------- unsigned int base 10.
2840541                   value_ui = va_arg (ap, unsigned int);
2840542                   s += uimaxtoa_fill (value_ui, &string[s], 10, 0,
```

```
2840543                                 alignment, filler, remain);
2840544               f += 1;
2840545             }
2840546           else if (format[f] == 'o')
2840547             {
2840548               //--------------------------------- unsigned int base 8.
2840549               value_ui = va_arg (ap, unsigned int);
2840550               s += uimaxtoa_fill (value_ui, &string[s], 8, 0,
2840551                                   alignment, filler, remain);
2840552               f += 1;
2840553             }
2840554           else if (format[f] == 'x')
2840555             {
2840556               //--------------------------------- unsigned int base 16.
2840557               value_ui = va_arg (ap, unsigned int);
2840558               s += uimaxtoa_fill (value_ui, &string[s], 16, 0,
2840559                                   alignment, filler, remain);
2840560               f += 1;
2840561             }
2840562           else if (format[f] == 'X')
2840563             {
2840564               //--------------------------------- unsigned int base 16.
2840565               value_ui = va_arg (ap, unsigned int);
2840566               s += uimaxtoa_fill (value_ui, &string[s], 16, 1,
2840567                                   alignment, filler, remain);
2840568               f += 1;
2840569             }
2840570           else if (format[f] == 'c')
2840571             {
2840572               //--------------------------------------- unsigned char.
2840573               value_ui = va_arg (ap, unsigned int);
2840574               string[s] = (char) value_ui;
2840575               s += 1;
2840576               f += 1;
2840577             }
2840578           else if (format[f] == 's')
2840579             {
2840580               //---------------------------------------------- string.
2840581               value_cp = va_arg (ap, char *);
2840582               filler = ' ';
2840583
2840584               s += strtostr_fill (value_cp, &string[s], alignment,
2840585                                   filler, remain);
```

```
2840586                 f += 1;
2840587               }
2840588             else
2840589               {
2840590                 //------------------- unsupported or unknown specifier.
2840591                 ;
2840592               }
2840593             //-------------------------------------------------------------
2840594             // End of specifier.
2840595             //-------------------------------------------------------------
2840596             width_string[0]     = '\0';
2840597             precision_string[0] = '\0';
2840598
2840599             specifier           = 0;
2840600             specifier_flags     = 0;
2840601             specifier_width     = 0;
2840602             specifier_precision = 0;
2840603             specifier_type      = 0;
2840604
2840605             flag_plus           = 0;
2840606             flag_minus          = 0;
2840607             flag_space          = 0;
2840608             flag_alternate      = 0;
2840609             flag_zero           = 0;
2840610           }
2840611       }
2840612     string[s] = '\0';
2840613     return s;
2840614 }
2840615 //---------------------------------------------------------------------
2840616 // Static functions.
2840617 //---------------------------------------------------------------------
2840618 static size_t
2840619 uimaxtoa (uintmax_t integer, char *buffer, int base, int uppercase,
2840620           size_t size)
2840621 {
2840622     //-----------------------------------------------------------------
2840623     // Convert a maximum rank integer into a string.
2840624     //-----------------------------------------------------------------
2840625
2840626         uintmax_t  integer_copy = integer;
2840627            size_t  digits;
2840628               int  b;
```

```
2840629       unsigned char  remainder;
2840630
2840631       for (digits = 0; integer_copy > 0; digits++)
2840632         {
2840633           integer_copy = integer_copy / base;
2840634         }
2840635
2840636       if (buffer == NULL && integer == 0) return 1;
2840637       if (buffer == NULL && integer > 0)  return digits;
2840638
2840639       if (integer == 0)
2840640         {
2840641           buffer[0] = '0';
2840642           buffer[1] = '\0';
2840643           return 1;
2840644         }
2840645       //
2840646       // Fix the maximum number of digits.
2840647       //
2840648       if (size > 0 && digits > size) digits = size;
2840649       //
2840650       *(buffer + digits) = '\0';            // End of string.
2840651
2840652       for (b = digits - 1; integer != 0 && b >= 0; b--)
2840653         {
2840654           remainder = integer % base;
2840655           integer   = integer / base;
2840656
2840657           if (remainder <= 9)
2840658             {
2840659               *(buffer + b) = remainder + '0';
2840660             }
2840661           else
2840662             {
2840663               if (uppercase)
2840664                 {
2840665                   *(buffer + b) = remainder - 10 + 'A';
2840666                 }
2840667               else
2840668                 {
2840669                   *(buffer + b) = remainder - 10 + 'a';
2840670                 }
2840671             }
```

```
2840672       }
2840673     return digits;
2840674 }
2840675 //---------------------------------------------------------------------
2840676 static size_t
2840677 imaxtoa (intmax_t integer, char *buffer, int base, int uppercase,
2840678          size_t size)
2840679 {
2840680     //-----------------------------------------------------------------
2840681     // Convert a maximum rank integer with sign into a string.
2840682     //-----------------------------------------------------------------
2840683
2840684     if (integer >= 0)
2840685       {
2840686         return uimaxtoa (integer, buffer, base, uppercase, size);
2840687       }
2840688     //
2840689     // At this point, there is a negative number, less than zero.
2840690     //
2840691     if (buffer == NULL)
2840692       {
2840693         return uimaxtoa (-integer, NULL, base, uppercase, size) + 1;
2840694       }
2840695
2840696     *buffer = '-';         // The minus sign is needed at the beginning.
2840697     if (size == 1)
2840698       {
2840699         *(buffer + 1) = '\0';
2840700         return 1;
2840701       }
2840702     else
2840703       {
2840704         return uimaxtoa (-integer, buffer+1, base, uppercase, size-1)
2840705                + 1;
2840706       }
2840707 }
2840708 //---------------------------------------------------------------------
2840709 static size_t
2840710 simaxtoa (intmax_t integer, char *buffer, int base, int uppercase,
2840711           size_t size)
2840712 {
2840713     //-----------------------------------------------------------------
2840714     // Convert a maximum rank integer with sign into a string, placing
```

```
2840715       // the sign also if it is positive.
2840716       //-----------------------------------------------------------------
2840717
2840718       if (buffer == NULL && integer >= 0)
2840719         {
2840720           return uimaxtoa (integer, NULL, base, uppercase, size) + 1;
2840721         }
2840722
2840723       if (buffer == NULL && integer < 0)
2840724         {
2840725           return uimaxtoa (-integer, NULL, base, uppercase, size) + 1;
2840726         }
2840727       //
2840728       // At this point, `buffer' is different from NULL.
2840729       //
2840730       if (integer >= 0)
2840731         {
2840732           *buffer = '+';
2840733         }
2840734       else
2840735         {
2840736           *buffer = '-';
2840737         }
2840738
2840739       if (size == 1)
2840740         {
2840741           *(buffer + 1) = '\0';
2840742           return 1;
2840743         }
2840744
2840745       if (integer >= 0)
2840746         {
2840747           return uimaxtoa (integer, buffer+1, base, uppercase, size-1)
2840748                  + 1;
2840749         }
2840750       else
2840751         {
2840752           return uimaxtoa (-integer, buffer+1, base, uppercase, size-1)
2840753                  + 1;
2840754         }
2840755   }
2840756   //-----------------------------------------------------------------
2840757   static size_t
```

```
2840758 uimaxtoa_fill (uintmax_t integer, char *buffer, int base,
2840759                 int uppercase, int width, int filler, int max)
2840760 {
2840761     //-----------------------------------------------------------------
2840762     // Convert a maximum rank integer without sign into a string,
2840763     // takeing care of the alignment.
2840764     //-----------------------------------------------------------------
2840765
2840766     size_t size_i;
2840767     size_t size_f;
2840768
2840769     if (max < 0) return 0; // «max» deve essere un valore positivo.
2840770
2840771     size_i = uimaxtoa (integer, NULL, base, uppercase, 0);
2840772
2840773     if (width > 0 &&  max > 0 && width >  max) width =  max;
2840774     if (width < 0 && -max < 0 && width < -max) width = -max;
2840775
2840776     if (size_i > abs (width))
2840777       {
2840778         return uimaxtoa (integer, buffer, base, uppercase, abs (width));
2840779       }
2840780
2840781     if (width == 0 && max > 0)
2840782       {
2840783         return uimaxtoa (integer, buffer, base, uppercase, max);
2840784       }
2840785
2840786     if (width == 0)
2840787       {
2840788         return uimaxtoa (integer, buffer, base, uppercase, abs (width));
2840789       }
2840790     //
2840791     // size_i <= abs (width).
2840792     //
2840793     size_f = abs (width) - size_i;
2840794
2840795     if (width < 0)
2840796       {
2840797         // Left alignment.
2840798         uimaxtoa (integer, buffer, base, uppercase, 0);
2840799         memset (buffer + size_i, filler, size_f);
2840800       }
```

```
2840801     else
2840802       {
2840803         // Right alignment.
2840804         memset (buffer, filler, size_f);
2840805         uimaxtoa (integer, buffer + size_f, base, uppercase, 0);
2840806       }
2840807     *(buffer + abs (width)) = '\0';
2840808
2840809     return abs (width);
2840810 }
2840811 //---------------------------------------------------------------------
2840812 static size_t
2840813 imaxtoa_fill (intmax_t integer, char *buffer, int base,
2840814               int uppercase, int width, int filler, int max)
2840815 {
2840816     //-----------------------------------------------------------------
2840817     // Convert a maximum rank integer with sign into a string,
2840818     // takeing care of the alignment.
2840819     //-----------------------------------------------------------------
2840820
2840821     size_t size_i;
2840822     size_t size_f;
2840823
2840824     if (max < 0) return 0; // `max' must be a positive value.
2840825
2840826     size_i = imaxtoa (integer, NULL, base, uppercase, 0);
2840827
2840828     if (width > 0 &&  max > 0 && width >  max) width =  max;
2840829     if (width < 0 && -max < 0 && width < -max) width = -max;
2840830
2840831     if (size_i > abs (width))
2840832       {
2840833         return imaxtoa (integer, buffer, base, uppercase, abs (width));
2840834       }
2840835
2840836     if (width == 0 && max > 0)
2840837       {
2840838         return imaxtoa (integer, buffer, base, uppercase, max);
2840839       }
2840840
2840841     if (width == 0)
2840842       {
2840843         return imaxtoa (integer, buffer, base, uppercase, abs (width));
```

```
2840844       }
2840845
2840846     // size_i <= abs (width).
2840847
2840848     size_f = abs (width) - size_i;
2840849
2840850     if (width < 0)
2840851       {
2840852         // Left alignment.
2840853         imaxtoa (integer, buffer, base, uppercase, 0);
2840854         memset (buffer + size_i, filler, size_f);
2840855       }
2840856     else
2840857       {
2840858         // Right alignment.
2840859         memset (buffer, filler, size_f);
2840860         imaxtoa (integer, buffer + size_f, base, uppercase, 0);
2840861       }
2840862     *(buffer + abs (width)) = '\0';
2840863
2840864     return abs (width);
2840865 }
2840866 //---------------------------------------------------------------------
2840867 static size_t
2840868 simaxtoa_fill (intmax_t integer, char *buffer, int base,
2840869                int uppercase, int width, int filler, int max)
2840870 {
2840871     //-----------------------------------------------------------------
2840872     // Convert a maximum rank integer with sign into a string,
2840873     // placing the sign also if it is positive and takeing care of the
2840874     // alignment.
2840875     //-----------------------------------------------------------------
2840876
2840877     size_t size_i;
2840878     size_t size_f;
2840879
2840880     if (max < 0) return 0; // `max' must be a positive value.
2840881
2840882     size_i = simaxtoa (integer, NULL, base, uppercase, 0);
2840883
2840884     if (width > 0 &&  max > 0 && width >  max) width =  max;
2840885     if (width < 0 && -max < 0 && width < -max) width = -max;
2840886
```

```
2840887       if (size_i > abs (width))
2840888         {
2840889           return simaxtoa (integer, buffer, base, uppercase, abs (width));
2840890         }
2840891
2840892       if (width == 0 && max > 0)
2840893         {
2840894           return simaxtoa (integer, buffer, base, uppercase, max);
2840895         }
2840896
2840897       if (width == 0)
2840898         {
2840899           return simaxtoa (integer, buffer, base, uppercase, abs (width));
2840900         }
2840901       //
2840902       // size_i <= abs (width).
2840903       //
2840904       size_f = abs (width) - size_i;
2840905
2840906       if (width < 0)
2840907         {
2840908           // Left alignment.
2840909           simaxtoa (integer, buffer, base, uppercase, 0);
2840910           memset (buffer + size_i, filler, size_f);
2840911         }
2840912       else
2840913         {
2840914           // Right alignment.
2840915           memset (buffer, filler, size_f);
2840916           simaxtoa (integer, buffer + size_f, base, uppercase, 0);
2840917         }
2840918       *(buffer + abs (width)) = '\0';
2840919
2840920       return abs (width);
2840921   }
2840922   //------------------------------------------------------------------------
2840923   static size_t
2840924   strtostr_fill (char *string, char *buffer, int width, int filler,
2840925                  int max)
2840926   {
2840927       //------------------------------------------------------------------
2840928       // Transfer a string with care for the alignment.
2840929       //------------------------------------------------------------------
```

```
2840930
2840931       size_t size_s;
2840932       size_t size_f;
2840933
2840934       if (max < 0) return 0; // 'max' must be a positive value.
2840935
2840936       size_s = strlen (string);
2840937
2840938       if (width > 0 &&  max > 0 && width >  max) width =  max;
2840939       if (width < 0 && -max < 0 && width < -max) width = -max;
2840940
2840941       if (width != 0 && size_s > abs (width))
2840942         {
2840943           memcpy (buffer, string, abs (width));
2840944           buffer[width] = '\0';
2840945           return width;
2840946         }
2840947
2840948       if (width == 0 && max > 0 && size_s > max)
2840949         {
2840950           memcpy (buffer, string, max);
2840951           buffer[max] = '\0';
2840952           return max;
2840953         }
2840954
2840955       if (width == 0 && max > 0 && size_s < max)
2840956         {
2840957           memcpy (buffer, string, size_s);
2840958           buffer[size_s] = '\0';
2840959           return size_s;
2840960         }
2840961       //
2840962       // width =! 0
2840963       // size_s <= abs (width)
2840964       //
2840965       size_f = abs (width) - size_s;
2840966
2840967       if (width < 0)
2840968         {
2840969           // Right alignment.
2840970           memset (buffer, filler, size_f);
2840971           strncpy (buffer+size_f, string, size_s);
2840972         }
```

```
2840973     else
2840974       {
2840975         // Left alignment.
2840976         strncpy (buffer, string, size_s);
2840977         memset (buffer+size_s, filler, size_f);
2840978       }
2840979     *(buffer + abs (width)) = '\0';
2840980
2840981     return abs (width);
2840982 }
2840983
2840984
```

lib/stdio/vsprintf.c

Si veda la sezione u0.128.

```
2850001 #include <stdio.h>
2850002 #include <sys/os16.h>
2850003 //-------------------------------------------------------------------
2850004 int
2850005 vsprintf (char *string, char *restrict format, va_list arg)
2850006 {
2850007     return (vsnprintf (string, BUFSIZ, format, arg));
2850008 }
```

lib/stdio/vsscanf.c

Si veda la sezione u0.129.

```
2860001 #include <stdio.h>
2860002
2860003 //-------------------------------------------------------------------
2860004 int vfsscanf (FILE *restrict fp, const char *string,
2860005                const char *restrict format, va_list ap);
2860006 //-------------------------------------------------------------------
2860007 int
2860008 vsscanf (const char *string, const char *restrict format, va_list ap)
2860009 {
2860010     return (vfsscanf (NULL, string, format, ap));
2860011 }
```

```
2860012 //------------------------------------------------------------------------
```

os16: «lib/stdlib.h»

«

Si veda la sezione u0.2.

```
2870001 #ifndef _STDLIB_H
2870002 #define _STDLIB_H       1
2870003 //------------------------------------------------------------------------
2870004
2870005 #include <size_t.h>
2870006 #include <wchar_t.h>
2870007 #include <NULL.h>
2870008 #include <limits.h>
2870009 #include <const.h>
2870010 #include <restrict.h>
2870011 //------------------------------------------------------------------------
2870012 typedef struct {
2870013     int       quot;
2870014     int       rem;
2870015 } div_t;
2870016 //------------------------------------------------------------------------
2870017 typedef struct {
2870018     long int quot;
2870019     long int rem;
2870020 } ldiv_t;
2870021 //------------------------------------------------------------------------
2870022 typedef void (*atexit_t) (void);        // Non standard. [1]
2870023 //
2870024 // [1] The type 'atexit_t' is a pointer to a function for the "at exit"
2870025 //     procedure, with no parameters and returning void. With the
2870026 //     declaration of type 'atexit_t', the function prototype of
2870027 //     'atexit()' is easier to declare and to understand. Original
2870028 //     declaration is:
2870029 //
2870030 //              int atexit (void (*function) (void));
2870031 //
2870032 //------------------------------------------------------------------------
2870033 #define EXIT_FAILURE    1
2870034 #define EXIT_SUCCESS    0
2870035 #define RAND_MAX        INT_MAX
2870036 #define MB_CUR_MAX      ((size_t) MB_LEN_MAX)
```

```
2870037 //---------------------------------------------------------------------
2870038 void               _Exit    (int status);
2870039 void               abort    (void);
2870040 int                abs      (int j);
2870041 int                atexit   (atexit_t function);
2870042 int                atoi     (const char *string);
2870043 long int           atol     (const char *string);
2870044 //void            *bsearch  (const void *key, const void *base,
2870045 //                           size_t nmemb, size_t size,
2870046 //                           int (*compar) (const void *, const void *));
2870047 #define            calloc(b, s) (malloc ((b) * (s)))
2870048 div_t              div      (int numer, int denom);
2870049 void               exit     (int status);
2870050 void               free     (void *ptr);
2870051 char              *getenv   (const char *name);
2870052 long int           labs     (long int j);
2870053 ldiv_t             ldiv     (long int numer, long int denom);
2870054 void              *malloc   (size_t size);
2870055 int                putenv   (const char *string);
2870056 void               qsort    (void *base, size_t nmemb, size_t size,
2870057                              int (*compare) (const void *,
2870058                                              const void *));
2870059 int                rand     (void);
2870060 void              *realloc  (void *ptr, size_t size);
2870061 int                setenv   (const char *name, const char *value,
2870062                              int overwrite);
2870063 void               srand    (unsigned int seed);
2870064 long int           strtol   (const char *restrict string,
2870065                              char **restrict endptr, int base);
2870066 unsigned long int strtoul   (const char * restrict string,
2870067                              char ** restrict endptr, int base);
2870068 //int              system   (const char *string);
2870069 int                unsetenv (const char *name);
2870070
2870071 #endif
```

lib/stdlib/_Exit.c

«

Si veda la sezione u0.2.

```
2880001 #include <stdlib.h>
2880002 #include <sys/os16.h>
```

```
2880003 //------------------------------------------------------------------------
2880004 void
2880005 _Exit (int status)
2880006 {
2880007     sysmsg_exit_t msg;
2880008     //
2880009     // Only the low eight bit are returned.
2880010     //
2880011     msg.status = (status & 0xFF);
2880012     //
2880013     //
2880014     //
2880015     sys (SYS_EXIT, &msg, (sizeof msg));
2880016     //
2880017     // Should not return from system call, but if it does, loop
2880018     // forever:
2880019     //
2880020     while (1);
2880021 }
```

lib/stdlib/abort.c

«

Si veda la sezione u0.2.

```
2890001 #include <stdlib.h>
2890002 #include <sys/types.h>
2890003 #include <signal.h>
2890004 #include <unistd.h>
2890005 //------------------------------------------------------------------------
2890006 void
2890007 abort (void)
2890008 {
2890009     pid_t        pid;
2890010     sighandler_t sig_previous;
2890011     //
2890012     // Set 'SIGABRT' to a default action.
2890013     //
2890014     sig_previous = signal (SIGABRT, SIG_DFL);
2890015     //
2890016     // If the previous action was something different than symbolic
2890017     // ones, configure again the previous action.
2890018     //
```

```
2890019     if (sig_previous != SIG_DFL &&
2890020         sig_previous != SIG_IGN &&
2890021         sig_previous != SIG_ERR)
2890022       {
2890023         signal (SIGABRT, sig_previous);
2890024       }
2890025     //
2890026     // Get current process ID and sent the signal.
2890027     //
2890028     pid = getpid ();
2890029     kill (pid, SIGABRT);
2890030     //
2890031     // Second chance
2890032     //
2890033     for (;;)
2890034       {
2890035         signal (SIGABRT, SIG_DFL);
2890036         pid = getpid ();
2890037         kill (pid, SIGABRT);
2890038       }
2890039 }
```

lib/stdlib/abs.c

«

Si veda la sezione u0.3.

```
2900001 #include <stdlib.h>
2900002 //---------------------------------------------------------------------
2900003 int
2900004 abs (int j)
2900005 {
2900006     if (j < 0)
2900007       {
2900008         return -j;
2900009       }
2900010     else
2900011       {
2900012         return j;
2900013       }
2900014 }
```

«

Si veda la sezione u0.66.

```
2910001 #include <stdlib.h>
2910002 #include <string.h>
2910003 #include <errno.h>
2910004 #include <limits.h>
2910005 #include <stdio.h>
2910006 //------------------------------------------------------------------------
2910007 #define MEMORY_BLOCK_SIZE       1024
2910008 //------------------------------------------------------------------------
2910009 static char     _alloc_memory[LONG_BIT][MEMORY_BLOCK_SIZE];   // [1]
2910010 static size_t   _alloc_size[LONG_BIT];                        // [2]
2910011 static long int _alloc_map;                                   // [3]
2910012 //
2910013 // [1] Memory to be allocated.
2910014 // [2] Sizes allocated.
2910015 // [3] Memory block map. The memory map is made of a single integer and
2910016 //     the rightmost bit is the first memory block.
2910017 //------------------------------------------------------------------------
2910018 void *
2910019 malloc (size_t size)
2910020 {
2910021     size_t   size_free;   // Size free found that might be allocated.
2910022     int      m;           // Index inside '_alloc_memory[][]' table.
2910023     int      s;           // Start index for a free memory area.
2910024     long int mask;        // Mask to compare with '_alloc_map'.
2910025     long int alloc;       // New allocation map.
2910026     //
2910027     // Check for arguments.
2910028     //
2910029     if (size == 0)
2910030       {
2910031         return (NULL);
2910032       }
2910033     //
2910034     for (s = 0, m = 0; m < LONG_BIT; m++)
2910035       {
2910036         mask = 1;
2910037         mask <<= m;
2910038         //
2910039         if (_alloc_map & mask)
2910040           {
```

```
2910041               //
2910042               // The memory block is not free.
2910043               //
2910044               s         = m + 1;
2910045               size_free = 0;
2910046               alloc     = 0;
2910047             }
2910048           else
2910049             {
2910050               alloc     |= mask;
2910051               size_free += MEMORY_BLOCK_SIZE;
2910052             }
2910053           if (size_free >= size)
2910054             {
2910055               //
2910056               // Space found: update '_alloc_size[]' table, the map inside
2910057               // '_alloc_map' and return the memory address.
2910058               //
2910059               _alloc_size[s] = size_free;
2910060               _alloc_map    |= alloc;
2910061               return ((void *) &_alloc_memory[s][0]);
2910062             }
2910063         }
2910064       //
2910065       // No space left.
2910066       //
2910067       errset (ENOMEM);                    // Not enough space.
2910068       //
2910069       return (NULL);
2910070   }
2910071   //-------------------------------------------------------------------------
2910072   void
2910073   free (void *address)
2910074   {
2910075       size_t   size_free;  // Size to make free.
2910076       int      m;          // Index inside '_alloc_memory[][]' table.
2910077       int      s;          // Start index.
2910078       long int mask;       // Mask to compare with '_alloc_map'.
2910079       long int alloc;      // New allocation map.
2910080       //
2910081       // Check argument.
2910082       //
2910083       if (address == NULL)
```

```
2910084         {
2910085           return;
2910086         }
2910087       //
2910088       // Find the original allocated address inside '_alloc_memory[][]'
2910089       // table.
2910090       //
2910091       for (m = 0; m < LONG_BIT; m++)
2910092         {
2910093           if (address == (void *) &_alloc_memory[m][0])
2910094             {
2910095               //
2910096               // This is the right memory block.
2910097               //
2910098               if (_alloc_size[m] == 0)
2910099                 {
2910100                   //
2910101                   // The block found is not allocated.
2910102                   //
2910103                   return;
2910104                 }
2910105               else
2910106                 {
2910107                   //
2910108                   // Build the map of the memory to set free.
2910109                   //
2910110                   size_free = _alloc_size[m];
2910111                   for (alloc = 0, s = m;
2910112                        size_free > 0 && s < LONG_BIT;
2910113                        size_free -= MEMORY_BLOCK_SIZE, s++)
2910114                     {
2910115                       mask   = 1;
2910116                       mask <<= s;
2910117                       alloc |= mask;
2910118                     }
2910119                   //
2910120                   // Compare the map of memory to be freed with the
2910121                   // reality allocated one, then free the memory.
2910122                   //
2910123                   if ((_alloc_map & alloc) == alloc)
2910124                     {
2910125                       _alloc_map    &= ~alloc;
2910126                       _alloc_size[m] = 0;
```

```
                    return;
                  }
                //
                // The real map does not report the same amount of
                // allocated memory, so nothing is freed.
                //
                return;
              }
          }
      }
    //
    // Address not allocated.
    //
    return;
}
//---------------------------------------------------------------------------
void *
realloc (void *address, size_t size)
{
    char  *address_new;
    char  *address_old = (char *) address;
    size_t size_old    = 0;
    size_t size_new    = size;
    int    m;                   // Index inside the memory table;
    //
    // Check arguments.
    //
    if (size == 0)              return (NULL);
    if (address == NULL)        return (malloc (size));
    //
    // Locate original allocation.
    //
    for (m = 0; m < LONG_BIT; m++)
      {
        if (address_old == (char *) &_alloc_memory[m][0])
          {
            size_old = _alloc_size[m];
            break;
          }
      }
    //
    // Check if a valid size was found.
    //
```

```
2910170       if (size_old == 0)
2910171         {
2910172           //
2910173           // Address not found or size not valid.
2910174           //
2910175           return (NULL);
2910176         }
2910177       //
2910178       // Allocate the new memory.
2910179       //
2910180       address_new = malloc (size);
2910181       //
2910182       // Check allocation. If there is an error, the variable 'errno'
2910183       // is already updated by 'malloc()'.
2910184       //
2910185       if (address_new == NULL)
2910186         {
2910187           return (NULL);
2910188         }
2910189       //
2910190       // Copy old memory.
2910191       //
2910192       for (; size_old > 0 && size_new > 0;
2910193            size_old--, size_new--, address_new++, address_old++)
2910194         {
2910195          *address_new = *address_old;
2910196         }
2910197       //
2910198       // Free old memory.
2910199       //
2910200       free (address);
2910201       //
2910202       // Return the new address.
2910203       //
2910204       return (address_new);
2910205   }
2910206   //---------------------------------------------------------------------
```

«

Si veda la sezione u0.4.

```
2920001 #include <stdlib.h>
2920002 #include <stdio.h>
2920003 //-------------------------------------------------------------------
2920004 atexit_t _atexit_table[ATEXIT_MAX];
2920005 //-------------------------------------------------------------------
2920006 void
2920007 _atexit_setup (void)
2920008 {
2920009     int a;
2920010     //
2920011     for (a = 0; a < ATEXIT_MAX; a++)
2920012       {
2920013         _atexit_table[a] = NULL;
2920014       }
2920015 }
2920016 //-------------------------------------------------------------------
2920017 int
2920018 atexit (atexit_t function)
2920019 {
2920020     int a;
2920021     //
2920022     if (function == NULL)
2920023       {
2920024         return (-1);
2920025       }
2920026     //
2920027     for (a = 0; a < ATEXIT_MAX; a++)
2920028       {
2920029         if (_atexit_table[a] == NULL)
2920030           {
2920031             _atexit_table[a] = function;
2920032             return (0);
2920033           }
2920034       }
2920035     //
2920036     return (-1);
2920037 }
```

«

Si veda la sezione u0.5.

```
2930001  #include <stdlib.h>
2930002  #include <ctype.h>
2930003  //------------------------------------------------------------------
2930004  int
2930005  atoi (const char *string)
2930006  {
2930007      int i;
2930008      int sign = +1;
2930009      int number;
2930010      //
2930011      for (i = 0; isspace (string[i]); i++)
2930012        {
2930013          ;
2930014        }
2930015      //
2930016      if (string[i] == '+')
2930017        {
2930018          sign = +1;
2930019          i++;
2930020        }
2930021      else if (string[i] == '-')
2930022        {
2930023          sign = -1;
2930024          i++;
2930025        }
2930026      //
2930027      for (number = 0; isdigit (string[i]); i++)
2930028        {
2930029          number *= 10;
2930030          number += (string[i] - '0');
2930031        }
2930032      //
2930033      number *= sign;
2930034      //
2930035      return number;
2930036  }
```

«

Si veda la sezione u0.5.

```
2940001  #include <stdlib.h>
2940002  #include <ctype.h>
2940003  //---------------------------------------------------------------------
2940004  long int
2940005  atol (const char *string)
2940006  {
2940007      int       i;
2940008      int       sign = +1;
2940009      long int number;
2940010      //
2940011      for (i = 0; isspace (string[i]); i++)
2940012        {
2940013          ;
2940014        }
2940015      //
2940016      if (string[i] == '+')
2940017        {
2940018          sign = +1;
2940019          i++;
2940020        }
2940021      else if (string[i] == '-')
2940022        {
2940023          sign = -1;
2940024          i++;
2940025        }
2940026      //
2940027      for (number = 0; isdigit (string[i]); i++)
2940028        {
2940029          number *= 10;
2940030          number += (string[i] - '0');
2940031        }
2940032      //
2940033      number *= sign;
2940034      //
2940035      return number;
2940036  }
```

lib/stdlib/div.c

«

Si veda la sezione u0.15.

```
2950001  #include <stdlib.h>
2950002  //---------------------------------------------------------------------
2950003  div_t
2950004  div (int numer, int denom)
2950005  {
2950006      div_t d;
2950007      d.quot = numer / denom;
2950008      d.rem = numer % denom;
2950009      return d;
2950010  }
```

lib/stdlib/environment.c

«

Si veda la sezione u0.1.

```
2960001  #include <stdlib.h>
2960002  #include <string.h>
2960003  //---------------------------------------------------------------------
2960004  // This file contains a non standard definition, related to the
2960005  // environment handling.
2960006  //
2960007  // The file `crt0.s', before calling the main function, calls the
2960008  // function `_environment_setup(), that is responsible for initializing
2960009  // the array `_environment_table[][]' and for copying the content
2960010  // of the environment, as it comes from the `exec()' system call.
2960011  //
2960012  // The pointers to the environment strings organised inside the
2960013  // array `_environment_table[][]', are also copied inside the
2960014  // array of pointers `_environment[]'.
2960015  //
2960016  // After all that is done, inside `crt0.s', the pointer to
2960017  // `_environment[]' is copied to the traditional variable `environ'
2960018  // and also to the previous value of the pointer variable `envp'.
2960019  //
2960020  // This way, applications will get the environment, but organised
2960021  // inside the table `_environment_table[][]'. So, functions like
2960022  // `getenv()' and `setenv()' do know where to look for.
2960023  //
2960024  // It is useful to notice that there is no prototype and no extern
```

```
2960025 // declaration inside the file <stdlib.h>, about this function
2960026 // and these arrays, because applications do not have to know about
2960027 // it.
2960028 //
2960029 // Please notice that 'environ' could be just the same as
2960030 // '_environment' here, but the common use puts 'environ' inside
2960031 // <unistd.h>, although for this implementation it should be better
2960032 // placed inside <stdlib.h>.
2960033 //
2960034 //---------------------------------------------------------------------
2960035 char  _environment_table[ARG_MAX/32][ARG_MAX/16];
2960036 char *_environment[ARG_MAX/32+1];
2960037 //---------------------------------------------------------------------
2960038 void
2960039 _environment_setup (char *envp[])
2960040 {
2960041     int e;
2960042     int s;
2960043     //
2960044     // Reset the '_environment_table[][]' array.
2960045     //
2960046     for (e = 0; e < ARG_MAX/32; e++)
2960047       {
2960048         for (s = 0; s < ARG_MAX/16; s++)
2960049           {
2960050             _environment_table[e][s] = 0;
2960051           }
2960052       }
2960053     //
2960054     // Set the '_environment[]' pointers. The final extra element must
2960055     // be a NULL pointer.
2960056     //
2960057     for (e = 0; e < ARG_MAX/32 ; e++)
2960058       {
2960059         _environment[e] = _environment_table[e];
2960060       }
2960061     _environment[ARG_MAX/32] = NULL;
2960062     //
2960063     // Copy the environment inside the array, but only if 'envp' is
2960064     // not NULL.
2960065     //
2960066     if (envp != NULL)
2960067       {
```

```
2960068           for (e = 0; envp[e] != NULL && e < ARG_MAX/32; e++)
2960069             {
2960070               strncpy (_environment_table[e], envp[e], (ARG_MAX/16)-1);
2960071             }
2960072         }
2960073   }
```

lib/stdlib/exit.c

«

Si veda la sezione u0.4.

```
2970001   #include <stdlib.h>
2970002   #include <stdio.h>
2970003   //-------------------------------------------------------------------------
2970004   extern atexit_t _atexit_table[];
2970005   //-------------------------------------------------------------------------
2970006   void
2970007   exit (int status)
2970008   {
2970009       int a;
2970010       //
2970011       // The "at exit" functions must be called in reverse order.
2970012       //
2970013       for (a = (ATEXIT_MAX - 1); a >= 0; a--)
2970014         {
2970015           if (_atexit_table[a] != NULL)
2970016             {
2970017               (*_atexit_table[a]) ();
2970018             }
2970019         }
2970020       //
2970021       // Now: really exit.
2970022       //
2970023       _Exit (status);
2970024       //
2970025       // Should not return from system call, but if it does, loop
2970026       // forever:
2970027       //
2970028       while (1);
2970029   }
```

«

Si veda la sezione u0.51.

```
2980001 #include <stdlib.h>
2980002 #include <string.h>
2980003 //--------------------------------------------------------------------
2980004 extern char *_environment[];
2980005 //--------------------------------------------------------------------
2980006 char *
2980007 getenv (const char *name)
2980008 {
2980009     int   e;         // First index: environment table items.
2980010     int   f;         // Second index: environment string scan.
2980011     char *value;     // Pointer to the environment value found.
2980012     //
2980013     // Check if the input is valid. No error is reported.
2980014     //
2980015     if (name == NULL || strlen (name) == 0)
2980016       {
2980017         return (NULL);
2980018       }
2980019     //
2980020     // Scan the environment table items, with index 'e'. The pointer
2980021     // 'value' is initialized to NULL. If the pointer 'value' gets a
2980022     // valid  pointer, the environment variable was found and a
2980023     // pointer to the beginning of its value is available.
2980024     //
2980025     for (value = NULL, e = 0; e < ARG_MAX/32; e++)
2980026       {
2980027         //
2980028         // Scan the string of the environment item, with index 'f'.
2980029         // The scan continue until 'name[f]' and '_environment[e][f]'
2980030         // are equal.
2980031         //
2980032         for (f = 0;
2980033              f < ARG_MAX/16-1 && name[f] == _environment[e][f];
2980034              f++)
2980035           {
2980036             ; // Just scan.
2980037           }
2980038         //
2980039         // At this point, 'name[f]' and '_environment[e][f]' are
2980040         // different: if 'name[f]' is zero the name string is
```

```
2980041           // terminated; if '_environment[e][f]' is also equal to '=',
2980042           // the environment item is corresponding to the requested name.
2980043           //
2980044           if (name[f] == 0 && _environment[e][f] == '=')
2980045             {
2980046               //
2980047               // The pointer to the beginning of the environment value is
2980048               // calculated, and the external loop exit.
2980049               //
2980050               value = &_environment[e][f+1];
2980051               break;
2980052             }
2980053         }
2980054       //
2980055       // The 'value' is returned: if it is still NULL, then, no
2980056       // environment variable with the requested name was found.
2980057       //
2980058       return (value);
2980059   }
```

lib/stdlib/labs.c

«

Si veda la sezione u0.3.

```
2990001   #include <stdlib.h>
2990002   //---------------------------------------------------------------------
2990003   long int
2990004   labs (long int j)
2990005   {
2990006       if (j < 0)
2990007         {
2990008           return -j;
2990009         }
2990010       else
2990011         {
2990012           return j;
2990013         }
2990014   }
```

lib/stdlib/ldiv.c

«

Si veda la sezione u0.15.

```
3000001  #include <stdlib.h>
3000002  //------------------------------------------------------------------------
3000003  ldiv_t
3000004  ldiv (long int numer, long int denom)
3000005  {
3000006      ldiv_t d;
3000007      d.quot = numer / denom;
3000008      d.rem = numer % denom;
3000009      return d;
3000010  }
```

lib/stdlib/putenv.c

«

Si veda la sezione u0.82.

```
3010001  #include <stdlib.h>
3010002  #include <string.h>
3010003  #include <errno.h>
3010004  //------------------------------------------------------------------------
3010005  extern char *_environment[];
3010006  //------------------------------------------------------------------------
3010007  int
3010008  putenv (const char *string)
3010009  {
3010010      int   e;         // First index: environment table items.
3010011      int   f;         // Second index: environment string scan.
3010012      //
3010013      // Check if the input is empty. No error is reported.
3010014      //
3010015      if (string == NULL || strlen (string) == 0)
3010016        {
3010017          return (0);
3010018        }
3010019      //
3010020      // Check if the input is valid: there must be a '=' sign.
3010021      // Error here is reported.
3010022      //
3010023      if (strchr (string, '=') == NULL)
3010024        {
```

```
3010025           errset(EINVAL);                   // Invalid argument.
3010026           return (-1);
3010027         }
3010028       //
3010029       // Scan the environment table items, with index 'e'. The intent is
3010030       // to find a previous environment variable with the same name.
3010031       //
3010032       for (e = 0; e < ARG_MAX/32; e++)
3010033         {
3010034           //
3010035           // Scan the string of the environment item, with index 'f'.
3010036           // The scan continue until 'string[f]' and '_environment[e][f]'
3010037           // are equal.
3010038           //
3010039           for (f = 0;
3010040                f < ARG_MAX/16-1 && string[f] == _environment[e][f];
3010041                f++)
3010042             {
3010043               ; // Just scan.
3010044             }
3010045           //
3010046           // At this point, 'string[f-1]' and '_environment[e][f-1]'
3010047           // should contain '='. If it is so, the environment is replaced.
3010048           //
3010049           if (string[f-1] == '=' && _environment[e][f-1] == '=')
3010050             {
3010051               //
3010052               // The environment item was found: now replace the pointer.
3010053               //
3010054               _environment[e] = string;
3010055               //
3010056               // Return.
3010057               //
3010058               return (0);
3010059             }
3010060         }
3010061       //
3010062       // The item was not found. Scan again for a free slot.
3010063       //
3010064       for (e = 0; e < ARG_MAX/32; e++)
3010065         {
3010066           if (_environment[e] == NULL || _environment[e][0] == 0)
3010067             {
```

```
3010068             //
3010069             // An empty item was found and the pointer will be
3010070             // replaced.
3010071             //
3010072             _environment[e] = string;
3010073             //
3010074             // Return.
3010075             //
3010076             return (0);
3010077           }
3010078       }
3010079     //
3010080     // Sorry: the empty slot was not found!
3010081     //
3010082     errset (ENOMEM);            // Not enough space.
3010083     return (-1);
3010084 }
```

lib/stdlib/qsort.c

«

Si veda la sezione u0.84.

```
3020001 #include <stdlib.h>
3020002 #include <string.h>
3020003 #include <errno.h>
3020004 //---------------------------------------------------------------------
3020005 static int  part (char *array, size_t size, int a, int z,
3020006                   int (*compare)(const void *, const void *));
3020007 static void sort (char *array, size_t size, int a, int z,
3020008                   int (*compare)(const void *, const void *));
3020009 //---------------------------------------------------------------------
3020010 void
3020011 qsort (void *base, size_t nmemb, size_t size,
3020012        int (*compare)(const void *, const void *))
3020013 {
3020014     if (size <= 1)
3020015       {
3020016         //
3020017         // There is nothing to sort!
3020018         //
3020019         return;
3020020       }
```

```
3020021     else
3020022       {
3020023         sort ((char *) base, size, 0, (int) (nmemb - 1), compare);
3020024       }
3020025 }
3020026 //-----------------------------------------------------------------------
3020027 static void
3020028 sort (char *array, size_t size, int a, int z,
3020029       int (*compare)(const void *, const void *))
3020030 {
3020031     int loc;
3020032     //
3020033     if (z > a)
3020034       {
3020035         loc = part (array, size, a, z, compare);
3020036         if (loc >= 0)
3020037           {
3020038             sort (array, size, a, loc-1, compare);
3020039             sort (array, size, loc+1, z, compare);
3020040           }
3020041       }
3020042 }
3020043
3020044 //-----------------------------------------------------------------------
3020045 static int
3020046 part (char *array, size_t size, int a, int z,
3020047       int (*compare)(const void *, const void *))
3020048 {
3020049     int   i;
3020050     int   loc;
3020051     char *swap;
3020052     //
3020053     if (z <= a)
3020054       {
3020055         errset (EUNKNOWN);       // Should never happen.
3020056         return (-1);
3020057       }
3020058     //
3020059     // Index 'i' after the first element; index 'loc' at the last
3020060     // position.
3020061     //
3020062     i   = a + 1;
3020063     loc = z;
```

```
3020064       //
3020065       // Prepare space in memory for element swap.
3020066       //
3020067       swap = malloc (size);
3020068       if (swap == NULL)
3020069         {
3020070           errset (ENOMEM);
3020071           return (-1);
3020072         }
3020073       //
3020074       // Loop as long as index 'loc' is higher than index 'i'.
3020075       // When index 'loc' is less or equal to index 'i',
3020076       // then, index 'loc' is the right position for the
3020077       // first element of the current piece of array.
3020078       //
3020079       for (;;)
3020080         {
3020081           //
3020082           // Index 'i' goes up...
3020083           //
3020084           for (;i < loc; i++)
3020085             {
3020086               if  (compare (&array[i*size], &array[a*size]) > 0)
3020087                 {
3020088                   break;
3020089                 }
3020090             }
3020091           //
3020092           // Index 'loc' gose down...
3020093           //
3020094           for (;; loc--)
3020095             {
3020096               if  (compare (&array[loc*size], &array[a*size]) <= 0)
3020097                 {
3020098                   break;
3020099                 }
3020100             }
3020101           //
3020102           // Swap elements related to index 'i' and 'loc'.
3020103           //
3020104           if (loc <= i)
3020105             {
3020106               //
```

```
3020107               // The array is completely scanned.
3020108               //
3020109               break;
3020110             }
3020111           else
3020112             {
3020113               memcpy (swap, &array[loc*size], size);
3020114               memcpy (&array[loc*size], &array[i*size], size);
3020115               memcpy (&array[i*size], swap, size);
3020116             }
3020117         }
3020118       //
3020119       // Swap the first element with the one related to the
3020120       // index 'loc'.
3020121       //
3020122       memcpy (swap, &array[loc*size], size);
3020123       memcpy (&array[loc*size], &array[a*size], size);
3020124       memcpy (&array[a*size], swap, size);
3020125       //
3020126       // Free the swap memory.
3020127       //
3020128       free (swap);
3020129       //
3020130       // Return the index 'loc'.
3020131       //
3020132       return (loc);
3020133   }
3020134
```

lib/stdlib/rand.c

«

Si veda la sezione u0.85.

```
3030001   #include <stdlib.h>
3030002   //---------------------------------------------------------------------
3030003   static unsigned int _srand = 1; // The '_srand' rank must be at least
3030004                                   // 'unsigned int' and must be able to
3030005                                   // represent the value 'RAND_MAX'.
3030006   //---------------------------------------------------------------------
3030007   int
3030008   rand (void)
3030009   {
```

```
3030010     _srand = _srand * 12345 + 123;
3030011     return _srand % ((unsigned int) RAND_MAX + 1);
3030012 }
3030013 //------------------------------------------------------------------------
3030014 void
3030015 srand (unsigned int seed)
3030016 {
3030017     _srand = seed;
3030018 }
```

lib/stdlib/setenv.c

«

Si veda la sezione u0.94.

```
3040001 #include <stdlib.h>
3040002 #include <string.h>
3040003 #include <errno.h>
3040004 //------------------------------------------------------------------------
3040005 extern char *_environment[];
3040006 extern char *_environment_table[];
3040007 //------------------------------------------------------------------------
3040008 int
3040009 setenv (const char *name, const char *value, int overwrite)
3040010 {
3040011     int   e;        // First index: environment table items.
3040012     int   f;        // Second index: environment string scan.
3040013     //
3040014     // Check if the input is empty. No error is reported.
3040015     //
3040016     if (name == NULL || strlen (name) == 0)
3040017       {
3040018         return (0);
3040019       }
3040020     //
3040021     // Check if the input is valid: error here is reported.
3040022     //
3040023     if (strchr (name, '=') != NULL)
3040024       {
3040025         errset(EINVAL);                 // Invalid argument.
3040026         return (-1);
3040027       }
3040028     //
```

```
3040029       // Check if the input is too big.
3040030       //
3040031       if ((strlen (name) + strlen (value) + 2) > ARG_MAX/16)
3040032         {
3040033           //
3040034           // The environment to be saved is bigger than the
3040035           // available string size, inside '_environment_table[]'.
3040036           //
3040037           errset (ENOMEM);                // Not enough space.
3040038           return (-1);
3040039         }
3040040       //
3040041       // Scan the environment table items, with index 'e'. The intent is
3040042       // to find a previous environment variable with the same name.
3040043       //
3040044       for (e = 0; e < ARG_MAX/32; e++)
3040045         {
3040046           //
3040047           // Scan the string of the environment item, with index 'f'.
3040048           // The scan continue until 'name[f]' and '_environment[e][f]'
3040049           // are equal.
3040050           //
3040051           for (f = 0;
3040052                f < ARG_MAX/16-1 && name[f] == _environment[e][f];
3040053                f++)
3040054             {
3040055               ; // Just scan.
3040056             }
3040057           //
3040058           // At this point, 'name[f]' and '_environment[e][f]' are
3040059           // different: if 'name[f]' is zero the name string is
3040060           // terminated; if '_environment[e][f]' is also equal to '=',
3040061           // the environment item is corresponding to the requested name.
3040062           //
3040063           if (name[f] == 0 && _environment[e][f] == '=')
3040064             {
3040065               //
3040066               // The environment item was found; if it can be overwritten,
3040067               // the write is done.
3040068               //
3040069               if (overwrite)
3040070                 {
3040071                   //
```

```
3040072                 // To be able to handle both 'setenv()' and 'putenv()',
3040073                 // before removing the item, it is fixed the pointer to
3040074                 // the global environment table.
3040075                 //
3040076                 _environment[e] = _environment_table[e];
3040077                 //
3040078                 // Now copy the new environment. The string size was
3040079                 // already checked.
3040080                 //
3040081                 strcpy (_environment[e], name);
3040082                 strcat (_environment[e], "=");
3040083                 strcat (_environment[e], value);
3040084                 //
3040085                 // Return.
3040086                 //
3040087                 return (0);
3040088               }
3040089             //
3040090             // Cannot overwrite!
3040091             //
3040092             errset (EUNKNOWN);
3040093             return (-1);
3040094           }
3040095       }
3040096     //
3040097     // The item was not found. Scan again for a free slot.
3040098     //
3040099     for (e = 0; e < ARG_MAX/32; e++)
3040100       {
3040101         if (_environment[e] == NULL || _environment[e][0] == 0)
3040102           {
3040103             //
3040104             // An empty item was found. To be able to handle both
3040105             // 'setenv()' and 'putenv()', it is fixed the pointer to
3040106             // the global environment table.
3040107             //
3040108             _environment[e] = _environment_table[e];
3040109             //
3040110             // Now copy the new environment. The string size was
3040111             // already checked.
3040112             //
3040113             strcpy (_environment[e], name);
3040114             strcat (_environment[e], "=");
```

```
3040115             strcat (_environment[e], value);
3040116             //
3040117             // Return.
3040118             //
3040119             return (0);
3040120           }
3040121       }
3040122     //
3040123     // Sorry: the empty slot was not found!
3040124     //
3040125     errset (ENOMEM);            // Not enough space.
3040126     return (-1);
3040127 }
```

lib/stdlib/strtol.c

«

Si veda la sezione u0.121.

```
3050001 #include <stdlib.h>
3050002 #include <ctype.h>
3050003 #include <errno.h>
3050004 #include <limits.h>
3050005 #include <stdbool.h>
3050006 //---------------------------------------------------------------------
3050007 #define isoctal(C)  ((int) (C >= '0' && C <= '7'))
3050008 //---------------------------------------------------------------------
3050009 long int
3050010 strtol (const char *restrict string, char **restrict endptr, int base)
3050011 {
3050012     int      i;
3050013     int      sign = +1;
3050014     long int number;
3050015     long int previous;
3050016     int      digit;
3050017     //
3050018     bool     flag_prefix_oct = 0;
3050019     bool     flag_prefix_exa = 0;
3050020     bool     flag_prefix_dec = 0;
3050021     //
3050022     // Check base and string.
3050023     //
3050024     if (base < 0
```

```
3050025       || base > 36
3050026       || base == 1                // With base 1 cannot do anything.
3050027       || string == NULL
3050028       || string[0] == 0)
3050029        {
3050030          if (endptr != NULL) *endptr = string;
3050031          errset (EINVAL);                // Invalid argument.
3050032          return ((long int) 0);
3050033        }
3050034      //
3050035      // Eat initial spaces.
3050036      //
3050037      for (i = 0; isspace (string[i]); i++)
3050038        {
3050039          ;
3050040        }
3050041      //
3050042      // Check sign.
3050043      //
3050044      if (string[i] == '+')
3050045        {
3050046          sign = +1;
3050047          i++;
3050048        }
3050049      else if (string[i] == '-')
3050050        {
3050051          sign = -1;
3050052          i++;
3050053        }
3050054      //
3050055      // Check for prefix.
3050056      //
3050057      if (string[i] == '0')
3050058        {
3050059          if (string[i+1] == 'x' || string[i+1] == 'X')
3050060            {
3050061              flag_prefix_exa = 1;
3050062            }
3050063          else if (isoctal (string[i+1]))
3050064            {
3050065              flag_prefix_oct = 1;
3050066            }
3050067          else
```

```
3050068             {
3050069               flag_prefix_dec = 1;
3050070             }
3050071         }
3050072       else if (isdigit (string[i]))
3050073         {
3050074           flag_prefix_dec = 1;
3050075         }
3050076       //
3050077       // Check compatibility with requested base.
3050078       //
3050079       if (flag_prefix_exa)
3050080         {
3050081           //
3050082           // At the moment, there is a zero and a 'x'. Might be
3050083           // exadecimal, or might be a number base 33 or more.
3050084           //
3050085           if (base == 0)
3050086             {
3050087               base = 16;
3050088             }
3050089           else if (base == 16)
3050090             {
3050091               ;   // Ok.
3050092             }
3050093           else if (base >= 33)
3050094             {
3050095               ;   // Ok.
3050096             }
3050097           else
3050098             {
3050099               //
3050100               // Incompatible sequence: only the initial zero is reported.
3050101               //
3050102               if (endptr != NULL) *endptr = &string[i+1];
3050103               return ((long int) 0);
3050104             }
3050105           //
3050106           // Move on, after the '0x' prefix.
3050107           //
3050108           i += 2;
3050109         }
3050110       //
```

```
3050111       if (flag_prefix_oct)
3050112         {
3050113           //
3050114           // There is a zero and a digit.
3050115           //
3050116           if (base == 0)
3050117             {
3050118               base = 8;
3050119             }
3050120           //
3050121           // Move on, after the '0' prefix.
3050122           //
3050123           i += 1;
3050124         }
3050125       //
3050126       if (flag_prefix_dec)
3050127         {
3050128           if (base == 0)
3050129             {
3050130               base = 10;
3050131             }
3050132         }
3050133       //
3050134       // Scan the string.
3050135       //
3050136       for (number = 0; string[i] != 0; i++)
3050137         {
3050138           if      (string[i] >= '0' && string[i] <= '9')
3050139             {
3050140               digit = string[i] - '0';
3050141             }
3050142           else if (string[i] >= 'A' && string[i] <= 'Z')
3050143             {
3050144               digit = string[i] - 'A' + 10;
3050145             }
3050146           else if (string[i] >= 'a' && string[i] <= 'z')
3050147             {
3050148               digit = string[i] - 'a' + 10;
3050149             }
3050150           else
3050151             {
3050152               //
3050153               // This is an out of range digit.
```

```
3050154               //
3050155               digit = 999;
3050156             }
3050157           //
3050158           // Give a sign to the digit.
3050159           //
3050160           digit *= sign;
3050161           //
3050162           // Compare with the base.
3050163           //
3050164           if (base > (digit * sign))
3050165             {
3050166               //
3050167               // Check if the current digit can be safely computed.
3050168               //
3050169               previous = number;
3050170               number  *= base;
3050171               number  += digit;
3050172               if (number / base != previous)
3050173                 {
3050174                   //
3050175                   // Out of range.
3050176                   //
3050177                   if (endptr != NULL) *endptr = &string[i+1];
3050178                   errset (ERANGE);        // Result too large.
3050179                   if (sign > 0)
3050180                     {
3050181                       return (LONG_MAX);
3050182                     }
3050183                   else
3050184                     {
3050185                       return (LONG_MIN);
3050186                     }
3050187                 }
3050188             }
3050189           else
3050190             {
3050191               if (endptr != NULL) *endptr = &string[i];
3050192               return (number);
3050193             }
3050194         }
3050195       //
3050196       // The string is finished.
```

```
3050197     //
3050198     if (endptr != NULL) *endptr = &string[i];
3050199     //
3050200     return (number);
3050201 }
```

lib/stdlib/strtoul.c

«

Si veda la sezione u0.121.

```
3060001 #include <stdlib.h>
3060002 #include <ctype.h>
3060003 #include <errno.h>
3060004 #include <limits.h>
3060005 //---------------------------------------------------------------------
3060006 // A really poor implementation. ,-(
3060007 //
3060008 unsigned long int
3060009 strtoul (const char *restrict string, char **restrict endptr, int base)
3060010 {
3060011     return ((unsigned long int) strtol (string, endptr, base));
3060012 }
```

lib/stdlib/unsetenv.c

«

Si veda la sezione u0.94.

```
3070001 #include <stdlib.h>
3070002 #include <string.h>
3070003 #include <errno.h>
3070004 //---------------------------------------------------------------------
3070005 extern char *_environment[];
3070006 extern char *_environment_table[];
3070007 //---------------------------------------------------------------------
3070008 int
3070009 unsetenv (const char *name)
3070010 {
3070011     int   e;        // First index: environment table items.
3070012     int   f;        // Second index: environment string scan.
3070013     //
3070014     // Check if the input is empty. No error is reported.
```

```
3070015       //
3070016       if (name == NULL || strlen (name) == 0)
3070017         {
3070018           return (0);
3070019         }
3070020       //
3070021       // Check if the input is valid: error here is reported.
3070022       //
3070023       if (strchr (name, '=') != NULL)
3070024         {
3070025           errset(EINVAL);                  // Invalid argument.
3070026           return (-1);
3070027         }
3070028       //
3070029       // Scan the environment table items, with index `e'.
3070030       //
3070031       for (e = 0; e < ARG_MAX/32; e++)
3070032         {
3070033           //
3070034           // Scan the string of the environment item, with index `f'.
3070035           // The scan continue until `name[f]' and `_environment[e][f]'
3070036           // are equal.
3070037           //
3070038           for (f = 0;
3070039                f < ARG_MAX/16-1 && name[f] == _environment[e][f];
3070040                f++)
3070041             {
3070042               ; // Just scan.
3070043             }
3070044           //
3070045           // At this point, `name[f]' and `_environment[e][f]' are
3070046           // different: if `name[f]' is zero the name string is
3070047           // terminated; if `_environment[e][f]' is also equal to `=',
3070048           // the environment item is corresponding to the requested name.
3070049           //
3070050           if (name[f] == 0 && _environment[e][f] == '=')
3070051             {
3070052               //
3070053               // The environment item was found and it have to be removed.
3070054               // To be able to handle both `setenv()' and `putenv()',
3070055               // before removing the item, it is fixed the pointer to
3070056               // the global environment table.
3070057               //
```

```
3070058                 _environment[e] = _environment_table[e];
3070059                 //
3070060                 // Now remove the environment item.
3070061                 //
3070062                 _environment[e][0] = 0;
3070063                 break;
3070064               }
3070065           }
3070066       //
3070067       // Work done fine.
3070068       //
3070069       return (0);
3070070   }
```

os16: «lib/string.h»

Si veda la sezione u0.2.

```
3080001   #ifndef _STRING_H
3080002   #define _STRING_H       1
3080003
3080004   #include <const.h>
3080005   #include <restrict.h>
3080006   #include <const.h>
3080007   #include <size_t.h>
3080008   #include <NULL.h>
3080009   //---------------------------------------------------------------------
3080010   void  *memccpy (void *restrict dst, const void *restrict org, int c,
3080011                   size_t n);
3080012   void  *memchr  (const void *memory, int c, size_t n);
3080013   int    memcmp  (const void *memory1, const void *memory2, size_t n);
3080014   void  *memcpy  (void *restrict dst, const void *restrict org, size_t n);
3080015   void  *memmove (void *dst, const void *org, size_t n);
3080016   void  *memset  (void *memory, int c, size_t n);
3080017   char  *strcat  (char *restrict dst, const char *restrict org);
3080018   char  *strchr  (const char *string, int c);
3080019   int    strcmp  (const char *string1, const char *string2);
3080020   int    strcoll (const char *string1, const char *string2);
3080021   char  *strcpy  (char *restrict dst, const char *restrict org);
3080022   size_t strcspn (const char *string, const char *reject);
3080023   char  *strdup  (const char *string);
3080024   char  *strerror (int errnum);
```

```
3080025 size_t strlen  (const char *string);
3080026 char  *strncat (char *restrict dst, const char *restrict org, size_t n);
3080027 int    strncmp (const char *string1, const char *string2, size_t n);
3080028 char  *strncpy (char *restrict dst, const char *restrict org, size_t n);
3080029 char  *strpbrk (const char *string, const char *accept);
3080030 char  *strrchr (const char *string, int c);
3080031 size_t strspn  (const char *string, const char *accept);
3080032 char  *strstr  (const char *string, const char *substring);
3080033 char  *strtok  (char *restrict string, const char *restrict delim);
3080034 size_t strxfrm (char *restrict dst, const char *restrict org, size_t n);
3080035 //------------------------------------------------------------------------
3080036
3080037
3080038
3080039 #endif
```

lib/string/memccpy.c

«

Si veda la sezione u0.67.

```
3090001 #include <string.h>
3090002 //------------------------------------------------------------------------
3090003 void *
3090004 memccpy (void *restrict dst, const void *restrict org, int c, size_t n)
3090005 {
3090006     char *d = (char *) dst;
3090007     char *o = (char *) org;
3090008     size_t i;
3090009     for (i = 0; n > 0 && i < n; i++)
3090010       {
3090011         d[i] = o[i];
3090012         if (d[i] == (char) c)
3090013           {
3090014             return ((void *) &d[i+1]);
3090015           }
3090016       }
3090017     return (NULL);
3090018 }
```

lib/string/memchr.c

«

Si veda la sezione u0.68.

```
3100001 #include <string.h>
3100002 //------------------------------------------------------------------------
3100003 void *
3100004 memchr (const void *memory, int c, size_t n)
3100005 {
3100006     char *m = (char *) memory;
3100007     size_t i;
3100008     for (i = 0; n > 0 && i < n; i++)
3100009       {
3100010         if (m[i] == (char) c)
3100011           {
3100012             return (void *) (m + i);
3100013           }
3100014       }
3100015     return NULL;
3100016 }
```

lib/string/memcmp.c

«

Si veda la sezione u0.69.

```
3110001 #include <string.h>
3110002 //------------------------------------------------------------------------
3110003 int
3110004 memcmp (const void *memory1, const void *memory2, size_t n)
3110005 {
3110006     char *a = (char *) memory1;
3110007     char *b = (char *) memory2;
3110008     size_t i;
3110009     for (i = 0; n > 0 && i < n; i++)
3110010       {
3110011         if (a[i] > b[i])
3110012           {
3110013             return 1;
3110014           }
3110015         else if (a[i] < b[i])
3110016           {
3110017             return -1;
3110018           }
```

```
3110019         }
3110020     return 0;
3110021 }
```

lib/string/memcpy.c

«

Si veda la sezione u0.70.

```
3120001 #include <string.h>
3120002 //---------------------------------------------------------------------
3120003 void *
3120004 memcpy (void *restrict dst, const void *restrict org, size_t n)
3120005 {
3120006     char *d = (char *) dst;
3120007     char *o = (char *) org;
3120008     size_t i;
3120009     for (i = 0; n > 0 && i < n; i++)
3120010       {
3120011         d[i] = o[i];
3120012       }
3120013     return dst;
3120014 }
```

lib/string/memmove.c

«

Si veda la sezione u0.71.

```
3130001 #include <string.h>
3130002 //---------------------------------------------------------------------
3130003 void *
3130004 memmove (void *dst, const void *org, size_t n)
3130005 {
3130006     char *d = (char *) dst;
3130007     char *o = (char *) org;
3130008     size_t i;
3130009     //
3130010     // Depending on the memory start locations, copy may be direct or
3130011     // reverse, to avoid overwriting before the relocation is done.
3130012     //
3130013     if (d < o)
3130014       {
```

```
3130015           for (i = 0; i < n; i++)
3130016             {
3130017               d[i] = o[i];
3130018             }
3130019         }
3130020       else if (d == o)
3130021         {
3130022           //
3130023           // Memory locations are already the same.
3130024           //
3130025           ;
3130026         }
3130027       else
3130028         {
3130029           for (i = n - 1; i >= 0; i--)
3130030             {
3130031               d[i] = o[i];
3130032             }
3130033         }
3130034       return dst;
3130035   }
```

lib/string/memset.c

«

Si veda la sezione u0.72.

```
3140001   #include <string.h>
3140002   //------------------------------------------------------------------------
3140003   void *
3140004   memset (void *memory, int c, size_t n)
3140005   {
3140006       char  *m = (char *) memory;
3140007       size_t i;
3140008       for (i = 0; n > 0 && i < n; i++)
3140009         {
3140010           m[i] = (char) c;
3140011         }
3140012       return memory;
3140013   }
```

lib/string/strcat.c

«

Si veda la sezione u0.104.

```
3150001 #include <string.h>
3150002 //------------------------------------------------------------------------
3150003 char *
3150004 strcat (char *restrict dst, const char *restrict org)
3150005 {
3150006     size_t i;
3150007     size_t j;
3150008     for (i = 0; dst[i] != 0; i++)
3150009       {
3150010         ; // Just look for the null character.
3150011       }
3150012     for (j = 0; org[j] != 0; i++, j++)
3150013       {
3150014         dst[i] = org[j];
3150015       }
3150016     dst[i] = 0;
3150017     return dst;
3150018 }
```

lib/string/strchr.c

«

Si veda la sezione u0.105.

```
3160001 #include <string.h>
3160002 //------------------------------------------------------------------------
3160003 char *
3160004 strchr (const char *string, int c)
3160005 {
3160006     size_t i;
3160007     for (i = 0; ; i++)
3160008       {
3160009         if (string[i] == (char) c)
3160010           {
3160011             return (char *) (string + i);
3160012           }
3160013         else if (string[i] == 0)
3160014           {
3160015             return NULL;
3160016           }
```

```
3160017         }
3160018 }
```

lib/string/strcmp.c

Si veda la sezione u0.106.

```
3170001 #include <string.h>
3170002 //------------------------------------------------------------------------
3170003 int
3170004 strcmp (const char *string1, const char *string2)
3170005 {
3170006     char *a = (char *) string1;
3170007     char *b = (char *) string2;
3170008     size_t i;
3170009     for (i = 0; ; i++)
3170010       {
3170011         if (a[i] > b[i])
3170012           {
3170013             return 1;
3170014           }
3170015         else if (a[i] < b[i])
3170016           {
3170017             return -1;
3170018           }
3170019         else if (a[i] == 0 && b[i] == 0)
3170020           {
3170021             return 0;
3170022           }
3170023       }
3170024 }
```

lib/string/strcoll.c

Si veda la sezione u0.106.

```
3180001 #include <string.h>
3180002 //------------------------------------------------------------------------
3180003 int
3180004 strcoll (const char *string1, const char *string2)
3180005 {
```

```
3180006         return (strcmp (string1, string2));
3180007     }
```

lib/string/strcpy.c

«

Si veda la sezione u0.108.

```
3190001     #include <string.h>
3190002     //------------------------------------------------------------------------
3190003     char *
3190004     strcpy (char *restrict dst, const char *restrict org)
3190005     {
3190006         size_t i;
3190007         for (i = 0; org[i] != 0; i++)
3190008           {
3190009             dst[i] = org[i];
3190010           }
3190011         dst[i] = 0;
3190012         return dst;
3190013     }
```

lib/string/strcspn.c

«

Si veda la sezione u0.118.

```
3200001     #include <string.h>
3200002     //------------------------------------------------------------------------
3200003     size_t
3200004     strcspn (const char *string, const char *reject)
3200005     {
3200006         size_t i;
3200007         size_t j;
3200008         int found;
3200009         for (i = 0; string[i] != 0; i++)
3200010           {
3200011             for (j = 0, found = 0; reject[j] != 0 || found; j++)
3200012               {
3200013                 if (string[i] == reject[j])
3200014                   {
3200015                     found = 1;
3200016                     break;
```

```
3200017                 }
3200018               }
3200019             if (found)
3200020               {
3200021                 break;
3200022               }
3200023           }
3200024       return i;
3200025   }
```

lib/string/strdup.c

«

Si veda la sezione u0.110.

```
3210001   #include <string.h>
3210002   #include <stdlib.h>
3210003   #include <errno.h>
3210004   //-----------------------------------------------------------------------
3210005   char *
3210006   strdup  (const char *string)
3210007   {
3210008       size_t size;
3210009       char  *copy;
3210010       //
3210011       // Get string size: must be added 1, to count the termination null
3210012       // character.
3210013       //
3210014       size = strlen (string) + 1;
3210015       //
3210016       copy = malloc (size);
3210017       //
3210018       if (copy == NULL)
3210019         {
3210020           errset (ENOMEM);         // Not enough memory.
3210021           return (NULL);
3210022         }
3210023       //
3210024       strcpy (copy, string);
3210025       //
3210026       return (copy);
3210027   }
```

«

Si veda la sezione u0.111.

```
3220001 #include <string.h>
3220002 #include <errno.h>
3220003 //------------------------------------------------------------------------
3220004 #define ERROR_MAX 100
3220005 //------------------------------------------------------------------------
3220006 char *
3220007 strerror (int errnum)
3220008 {
3220009     static char *err[ERROR_MAX];
3220010     //
3220011     err[0]                                  = "No error";
3220012     err[E2BIG]                              = TEXT_E2BIG;
3220013     err[EACCES]                             = TEXT_EACCES;
3220014     err[EADDRINUSE]                         = TEXT_EADDRINUSE;
3220015     err[EADDRNOTAVAIL]                      = TEXT_EADDRNOTAVAIL;
3220016     err[EAFNOSUPPORT]                       = TEXT_EAFNOSUPPORT;
3220017     err[EAGAIN]                             = TEXT_EAGAIN;
3220018     err[EALREADY]                           = TEXT_EALREADY;
3220019     err[EBADF]                              = TEXT_EBADF;
3220020     err[EBADMSG]                            = TEXT_EBADMSG;
3220021     err[EBUSY]                              = TEXT_EBUSY;
3220022     err[ECANCELED]                          = TEXT_ECANCELED;
3220023     err[ECHILD]                             = TEXT_ECHILD;
3220024     err[ECONNABORTED]                       = TEXT_ECONNABORTED;
3220025     err[ECONNREFUSED]                       = TEXT_ECONNREFUSED;
3220026     err[ECONNRESET]                         = TEXT_ECONNRESET;
3220027     err[EDEADLK]                            = TEXT_EDEADLK;
3220028     err[EDESTADDRREQ]                       = TEXT_EDESTADDRREQ;
3220029     err[EDOM]                               = TEXT_EDOM;
3220030     err[EDQUOT]                             = TEXT_EDQUOT;
3220031     err[EEXIST]                             = TEXT_EEXIST;
3220032     err[EFAULT]                             = TEXT_EFAULT;
3220033     err[EFBIG]                              = TEXT_EFBIG;
3220034     err[EHOSTUNREACH]                       = TEXT_EHOSTUNREACH;
3220035     err[EIDRM]                              = TEXT_EIDRM;
3220036     err[EILSEQ]                             = TEXT_EILSEQ;
3220037     err[EINPROGRESS]                        = TEXT_EINPROGRESS;
3220038     err[EINTR]                              = TEXT_EINTR;
3220039     err[EINVAL]                             = TEXT_EINVAL;
3220040     err[EIO]                                = TEXT_EIO;
```

```
3220041     err[EISCONN]                        = TEXT_EISCONN;
3220042     err[EISDIR]                         = TEXT_EISDIR;
3220043     err[ELOOP]                          = TEXT_ELOOP;
3220044     err[EMFILE]                         = TEXT_EMFILE;
3220045     err[EMLINK]                         = TEXT_EMLINK;
3220046     err[EMSGSIZE]                       = TEXT_EMSGSIZE;
3220047     err[EMULTIHOP]                      = TEXT_EMULTIHOP;
3220048     err[ENAMETOOLONG]                   = TEXT_ENAMETOOLONG;
3220049     err[ENETDOWN]                       = TEXT_ENETDOWN;
3220050     err[ENETRESET]                      = TEXT_ENETRESET;
3220051     err[ENETUNREACH]                    = TEXT_ENETUNREACH;
3220052     err[ENFILE]                         = TEXT_ENFILE;
3220053     err[ENOBUFS]                        = TEXT_ENOBUFS;
3220054     err[ENODATA]                        = TEXT_ENODATA;
3220055     err[ENODEV]                         = TEXT_ENODEV;
3220056     err[ENOENT]                         = TEXT_ENOENT;
3220057     err[ENOEXEC]                        = TEXT_ENOEXEC;
3220058     err[ENOLCK]                         = TEXT_ENOLCK;
3220059     err[ENOLINK]                        = TEXT_ENOLINK;
3220060     err[ENOMEM]                         = TEXT_ENOMEM;
3220061     err[ENOMSG]                         = TEXT_ENOMSG;
3220062     err[ENOPROTOOPT]                    = TEXT_ENOPROTOOPT;
3220063     err[ENOSPC]                         = TEXT_ENOSPC;
3220064     err[ENOSR]                          = TEXT_ENOSR;
3220065     err[ENOSTR]                         = TEXT_ENOSTR;
3220066     err[ENOSYS]                         = TEXT_ENOSYS;
3220067     err[ENOTCONN]                       = TEXT_ENOTCONN;
3220068     err[ENOTDIR]                        = TEXT_ENOTDIR;
3220069     err[ENOTEMPTY]                      = TEXT_ENOTEMPTY;
3220070     err[ENOTSOCK]                       = TEXT_ENOTSOCK;
3220071     err[ENOTSUP]                        = TEXT_ENOTSUP;
3220072     err[ENOTTY]                         = TEXT_ENOTTY;
3220073     err[ENXIO]                          = TEXT_ENXIO;
3220074     err[EOPNOTSUPP]                     = TEXT_EOPNOTSUPP;
3220075     err[EOVERFLOW]                      = TEXT_EOVERFLOW;
3220076     err[EPERM]                          = TEXT_EPERM;
3220077     err[EPIPE]                          = TEXT_EPIPE;
3220078     err[EPROTO]                         = TEXT_EPROTO;
3220079     err[EPROTONOSUPPORT]                = TEXT_EPROTONOSUPPORT;
3220080     err[EPROTOTYPE]                     = TEXT_EPROTOTYPE;
3220081     err[ERANGE]                         = TEXT_ERANGE;
3220082     err[EROFS]                          = TEXT_EROFS;
3220083     err[ESPIPE]                         = TEXT_ESPIPE;
```

```
3220084     err[ESRCH]                          = TEXT_ESRCH;
3220085     err[ESTALE]                         = TEXT_ESTALE;
3220086     err[ETIME]                          = TEXT_ETIME;
3220087     err[ETIMEDOUT]                      = TEXT_ETIMEDOUT;
3220088     err[ETXTBSY]                        = TEXT_ETXTBSY;
3220089     err[EWOULDBLOCK]                    = TEXT_EWOULDBLOCK;
3220090     err[EXDEV]                          = TEXT_EXDEV;
3220091     err[E_FILE_TYPE]                    = TEXT_E_FILE_TYPE;
3220092     err[E_ROOT_INODE_NOT_CACHED]        = TEXT_E_ROOT_INODE_NOT_CACHED;
3220093     err[E_CANNOT_READ_SUPERBLOCK]       = TEXT_E_CANNOT_READ_SUPERBLOCK;
3220094     err[E_MAP_INODE_TOO_BIG]            = TEXT_E_MAP_INODE_TOO_BIG;
3220095     err[E_MAP_ZONE_TOO_BIG]             = TEXT_E_MAP_ZONE_TOO_BIG;
3220096     err[E_DATA_ZONE_TOO_BIG]            = TEXT_E_DATA_ZONE_TOO_BIG;
3220097     err[E_CANNOT_FIND_ROOT_DEVICE]      = TEXT_E_CANNOT_FIND_ROOT_DEVICE;
3220098     err[E_CANNOT_FIND_ROOT_INODE]       = TEXT_E_CANNOT_FIND_ROOT_INODE;
3220099     err[E_FILE_TYPE_UNSUPPORTED]        = TEXT_E_FILE_TYPE_UNSUPPORTED;
3220100     err[E_ENV_TOO_BIG]                  = TEXT_E_ENV_TOO_BIG;
3220101     err[E_LIMIT]                        = TEXT_E_LIMIT;
3220102     err[E_NOT_MOUNTED]                  = TEXT_E_NOT_MOUNTED;
3220103     err[E_NOT_IMPLEMENTED]              = TEXT_E_NOT_IMPLEMENTED;
3220104     //
3220105     if (errnum >= ERROR_MAX || errnum < 0)
3220106       {
3220107         return ("Unknown error");
3220108       }
3220109     //
3220110     return (err[errnum]);
3220111 }
```

lib/string/strlen.c

«

Si veda la sezione u0.112.

```
3230001 #include <string.h>
3230002 //---------------------------------------------------------------------
3230003 size_t
3230004 strlen (const char *string)
3230005 {
3230006     size_t i;
3230007     for (i = 0; string[i] != 0 ; i++)
3230008       {
3230009         ; // Just count.
```

```
3230010         }
3230011       return i;
3230012   }
```

lib/string/strncat.c

Si veda la sezione u0.104.

```
3240001   #include <string.h>
3240002   //------------------------------------------------------------------------
3240003   char *
3240004   strncat (char *restrict dst, const char *restrict org, size_t n)
3240005   {
3240006       size_t i;
3240007       size_t j;
3240008       for (i = 0; n > 0 && dst[i] != 0; i++)
3240009         {
3240010           ; // Just seek the null character.
3240011         }
3240012       for (j = 0; n > 0 && j < n && org[j] != 0; i++, j++)
3240013         {
3240014           dst[i] = org[j];
3240015         }
3240016       dst[i] = 0;
3240017       return dst;
3240018   }
```

lib/string/strncmp.c

Si veda la sezione u0.106.

```
3250001   #include <string.h>
3250002   //------------------------------------------------------------------------
3250003   int
3250004   strncmp (const char *string1, const char *string2, size_t n)
3250005   {
3250006       size_t i;
3250007       for (i = 0; i < n ; i++)
3250008         {
3250009           if (string1[i] > string2[i])
3250010             {
```

```
3250011             return 1;
3250012           }
3250013         else if (string1[i] < string2[i])
3250014           {
3250015             return -1;
3250016           }
3250017         else if (string1[i] == 0 && string2[i] == 0)
3250018           {
3250019             return 0;
3250020           }
3250021       }
3250022     return 0;
3250023 }
```

lib/string/strncpy.c

«

Si veda la sezione u0.108.

```
3260001 #include <string.h>
3260002 //---------------------------------------------------------------------
3260003 char *
3260004 strncpy (char *restrict dst, const char *restrict org, size_t n)
3260005 {
3260006     size_t i;
3260007     for (i = 0; n > 0 && i < n && org[i] != 0; i++)
3260008       {
3260009         dst[i] = org[i];
3260010       }
3260011     for ( ; n > 0 && i < n; i++)
3260012       {
3260013         dst[i] = 0;
3260014       }
3260015     return dst;
3260016 }
```

lib/string/strpbrk.c

«

Si veda la sezione u0.116.

```
3270001 #include <string.h>
3270002 //------------------------------------------------------------------------
3270003 char *
3270004 strpbrk (const char *string, const char *accept)
3270005 {
3270006     size_t i;
3270007     size_t j;
3270008     for (i = 0; string[i] != 0; i++)
3270009       {
3270010         for (j = 0; accept[j] != 0; j++)
3270011           {
3270012             if (string[i] == accept[j])
3270013               {
3270014                 return (string + i);
3270015               }
3270016           }
3270017       }
3270018     return NULL;
3270019 }
```

lib/string/strrchr.c

«

Si veda la sezione u0.105.

```
3280001 #include <string.h>
3280002 //------------------------------------------------------------------------
3280003 char *
3280004 strrchr (const char *string, int c)
3280005 {
3280006     int i;
3280007     for (i = strlen (string); i >= 0; i--)
3280008       {
3280009         if (string[i] == (char) c)
3280010           {
3280011             break;
3280012           }
3280013       }
3280014     if (i < 0)
3280015       {
```

```
3280016           return NULL;
3280017         }
3280018       else
3280019         {
3280020           return (string + i);
3280021         }
3280022   }
```

lib/string/strspn.c

«

Si veda la sezione u0.118.

```
3290001   #include <string.h>
3290002   //------------------------------------------------------------------------
3290003   size_t
3290004   strspn (const char *string, const char *accept)
3290005   {
3290006       size_t i;
3290007       size_t j;
3290008       int found;
3290009       for (i = 0; string[i] != 0; i++)
3290010         {
3290011           for (j = 0, found = 0; accept[j] != 0; j++)
3290012             {
3290013               if (string[i] == accept[j])
3290014                 {
3290015                   found = 1;
3290016                   break;
3290017                 }
3290018             }
3290019           if (!found)
3290020             {
3290021               break;
3290022             }
3290023         }
3290024       return i;
3290025   }
```

«

Si veda la sezione u0.119.

```
3300001 #include <string.h>
3300002 //---------------------------------------------------------------------
3300003 char *
3300004 strstr (const char *string, const char *substring)
3300005 {
3300006     size_t i;
3300007     size_t j;
3300008     size_t k;
3300009     int found;
3300010     if (substring[0] == 0)
3300011       {
3300012         return (char *) string;
3300013       }
3300014     for (i = 0, j = 0, found = 0; string[i] != 0; i++)
3300015       {
3300016         if (string[i] == substring[0])
3300017           {
3300018             for (k = i, j = 0;
3300019                  string[k] == substring[j] &&
3300020                  string[k] != 0            &&
3300021                  substring[j] != 0;
3300022                  j++, k++)
3300023               {
3300024                 ;
3300025               }
3300026             if (substring[j] == 0)
3300027               {
3300028                 found = 1;
3300029               }
3300030           }
3300031         if (found)
3300032           {
3300033             return (char *) (string + i);
3300034           }
3300035       }
3300036     return NULL;
3300037 }
```

«

Si veda la sezione u0.120.

```
3310001 #include <string.h>
3310002 //-----------------------------------------------------------------------
3310003 char *
3310004 strtok (char *restrict string, const char *restrict delim)
3310005 {
3310006     static char *next = NULL;
3310007     size_t i = 0;
3310008     size_t j;
3310009     int found_token;
3310010     int found_delim;
3310011     //
3310012     // If the string received a the first parameter is a null pointer,
3310013     // the static pointer is used. But if it is already NULL,
3310014     // the scan cannot start.
3310015     //
3310016     if (string == NULL)
3310017       {
3310018         if (next == NULL)
3310019           {
3310020             return NULL;
3310021           }
3310022         else
3310023           {
3310024             string = next;
3310025           }
3310026       }
3310027     //
3310028     // If the string received as the first parameter is empty, the scan
3310029     // cannot start.
3310030     //
3310031     if (string[0] == 0)
3310032       {
3310033         next = NULL;
3310034         return NULL;
3310035       }
3310036     else
3310037       {
3310038         if (delim[0] == 0)
3310039           {
3310040             return string;
```

```
3310041             }
3310042         }
3310043       //
3310044       // Find the next token.
3310045       //
3310046       for (i = 0, found_token = 0, j = 0;
3310047            string[i] != 0 && (!found_token); i++)
3310048         {
3310049           //
3310050           // Look inside delimiters.
3310051           //
3310052           for (j = 0, found_delim = 0; delim[j] != 0; j++)
3310053             {
3310054               if (string[i] == delim[j])
3310055                 {
3310056                   found_delim = 1;
3310057                 }
3310058             }
3310059           //
3310060           // If current character inside the string is not a delimiter,
3310061           // it is the start of a new token.
3310062           //
3310063           if (!found_delim)
3310064             {
3310065               found_token = 1;
3310066               break;
3310067             }
3310068         }
3310069       //
3310070       // If a token was found, the pointer is updated.
3310071       // If otherwise the token is not found, this means that
3310072       // there are no more.
3310073       //
3310074       if (found_token)
3310075         {
3310076           string += i;
3310077         }
3310078       else
3310079         {
3310080           next = NULL;
3310081           return NULL;
3310082         }
3310083       //
```

```
3310084       // Find the end of the token.
3310085       //
3310086       for (i = 0, found_delim = 0; string[i] != 0; i++)
3310087         {
3310088           for (j = 0; delim[j] != 0; j++)
3310089             {
3310090               if (string[i] == delim[j])
3310091                 {
3310092                   found_delim = 1;
3310093                   break;
3310094                 }
3310095             }
3310096           if (found_delim)
3310097             {
3310098               break;
3310099             }
3310100         }
3310101       //
3310102       // If a delimiter was found, the corresponding character must be
3310103       // reset to zero. If otherwise the string is terminated, the
3310104       // scan is terminated.
3310105       //
3310106       if (found_delim)
3310107         {
3310108           string[i] = 0;
3310109           next = &string[i+1];
3310110         }
3310111       else
3310112         {
3310113           next = NULL;
3310114         }
3310115       //
3310116       // At this point, the current string represent the token found.
3310117       //
3310118       return string;
3310119   }
```

«

Si veda la sezione u0.123.

```
3320001  #include <string.h>
3320002  //-----------------------------------------------------------------------
3320003  size_t
3320004  strxfrm (char *restrict dst, const char *restrict org, size_t n)
3320005  {
3320006      size_t i;
3320007      if (n == 0 && dst == NULL)
3320008        {
3320009          return strlen (org);
3320010        }
3320011      else
3320012        {
3320013          for (i = 0; i < n ; i++)
3320014            {
3320015              dst[i] = org[i];
3320016              if (org[i] == 0)
3320017                {
3320018                  break;
3320019                }
3320020            }
3320021          return i;
3320022        }
3320023  }
```

os16: «lib/sys/os16.h»

«

Si veda la sezione u0.2.

```
3330001  #ifndef _SYS_OS16_H
3330002  #define _SYS_OS16_H      1
3330003  //-----------------------------------------------------------------------
3330004  // This file contains all the declarations that don't have a better
3330005  // place inside standard headers files. Even declarations related to
3330006  // device numbers and system calls is contained here.
3330007  //-----------------------------------------------------------------------
3330008  // Please remember that system calls should never be used (called)
3330009  // inside the kernel code, because system calls cannot be nested for
3330010  // the os16 simple architecture!
3330011  // If a particular function is necessary inside the kernel, that usually
```

```
3330012 // is made by a system call, an appropriate k_...() function must be
3330013 // made, to avoid the problem.
3330014 //------------------------------------------------------------------------
3330015
3330016 #include <sys/types.h>
3330017 #include <sys/stat.h>
3330018 #include <stdint.h>
3330019 #include <signal.h>
3330020 #include <limits.h>
3330021 #include <stdio.h>
3330022 #include <stdint.h>
3330023 #include <stddef.h>
3330024 #include <const.h>
3330025 #include <restrict.h>
3330026 #include <stdarg.h>
3330027 //------------------------------------------------------------------------
3330028 // Device numbers.
3330029 //------------------------------------------------------------------------
3330030 #define DEV_UNDEFINED_MAJOR     0x00
3330031 #define DEV_UNDEFINED           0x0000
3330032 #define DEV_MEM_MAJOR           0x01
3330033 #define DEV_MEM                 0x0101
3330034 #define DEV_NULL                0x0102
3330035 #define DEV_PORT                0x0103
3330036 #define DEV_ZERO                0x0104
3330037 #define DEV_TTY_MAJOR           0x02
3330038 #define DEV_TTY                 0x0200
3330039 #define DEV_DSK_MAJOR           0x03
3330040 #define DEV_DSK0                0x0300
3330041 #define DEV_DSK1                0x0301
3330042 #define DEV_DSK2                0x0302
3330043 #define DEV_DSK3                0x0303
3330044 #define DEV_KMEM_MAJOR          0x04
3330045 #define DEV_KMEM_PS             0x0401
3330046 #define DEV_KMEM_MMP            0x0402
3330047 #define DEV_KMEM_SB             0x0403
3330048 #define DEV_KMEM_INODE          0x0404
3330049 #define DEV_KMEM_FILE           0x0405
3330050 #define DEV_CONSOLE_MAJOR       0x05
3330051 #define DEV_CONSOLE             0x05FF
3330052 #define DEV_CONSOLE0            0x0500
3330053 #define DEV_CONSOLE1            0x0501
3330054 #define DEV_CONSOLE2            0x0502
```

```
3330055 #define DEV_CONSOLE3            0x0503
3330056 #define DEV_CONSOLE4            0x0504
3330057 //-----------------------------------------------------------------------
3330058 // Current segments.
3330059 //-----------------------------------------------------------------------
3330060 uint16_t _seg_i (void);
3330061 uint16_t _seg_d (void);
3330062 uint16_t _cs    (void);
3330063 uint16_t _ds    (void);
3330064 uint16_t _ss    (void);
3330065 uint16_t _es    (void);
3330066 uint16_t _sp    (void);
3330067 uint16_t _bp    (void);
3330068 #define seg_i() ((unsigned int) _seg_i ())
3330069 #define seg_d() ((unsigned int) _seg_d ())
3330070 #define cs()    ((unsigned int) _cs ())
3330071 #define ds()    ((unsigned int) _ds ())
3330072 #define ss()    ((unsigned int) _ss ())
3330073 #define es()    ((unsigned int) _es ())
3330074 #define sp()    ((unsigned int) _sp ())
3330075 #define bp()    ((unsigned int) _bp ())
3330076 //-----------------------------------------------------------------------
3330077 #define min(a, b) (a < b ? a : b)
3330078 #define max(a, b) (a > b ? a : b)
3330079 //-----------------------------------------------------------------------
3330080 #define INPUT_LINE_HIDDEN 0
3330081 #define INPUT_LINE_ECHO   1
3330082 #define INPUT_LINE_STARS  2
3330083 //-----------------------------------------------------------------------
3330084 #define MOUNT_DEFAULT     0  // Default mount options.
3330085 #define MOUNT_RO          1  // Read only mount option.
3330086 //-----------------------------------------------------------------------
3330087 #define SYS_0                    0  // Nothing to do.
3330088 #define SYS_CHDIR                1
3330089 #define SYS_CHMOD                2
3330090 #define SYS_CLOCK                3
3330091 #define SYS_CLOSE                4
3330092 #define SYS_EXEC                 5
3330093 #define SYS_EXIT                 6  // [1] see below.
3330094 #define SYS_FCHMOD               7
3330095 #define SYS_FORK                 8
3330096 #define SYS_FSTAT                9
3330097 #define SYS_KILL                10
```

```
3330098 #define SYS_LSEEK               11
3330099 #define SYS_MKDIR               12
3330100 #define SYS_MKNOD               13
3330101 #define SYS_MOUNT               14
3330102 #define SYS_OPEN                15
3330103 #define SYS_PGRP                16
3330104 #define SYS_READ                17
3330105 #define SYS_SETEUID             18
3330106 #define SYS_SETUID              19
3330107 #define SYS_SIGNAL              20
3330108 #define SYS_SLEEP               21
3330109 #define SYS_STAT                22
3330110 #define SYS_TIME                23
3330111 #define SYS_UAREA               24
3330112 #define SYS_UMASK               25
3330113 #define SYS_UMOUNT              26
3330114 #define SYS_WAIT                27
3330115 #define SYS_WRITE               28
3330116 #define SYS_ZPCHAR              29  // [2] see below.
3330117 #define SYS_ZPSTRING            30  // [2]
3330118 #define SYS_CHOWN               31
3330119 #define SYS_DUP                 33
3330120 #define SYS_DUP2                34
3330121 #define SYS_LINK                35
3330122 #define SYS_UNLINK              36
3330123 #define SYS_FCNTL               37
3330124 #define SYS_STIME               38
3330125 #define SYS_FCHOWN              39
3330126 //
3330127 // [1] The files 'crt0...' need to know the value used for the
3330128 //     exit system call. If this value is modified, all the file
3330129 //     'crt0...' have also to be modified the same way.
3330130 //
3330131 // [2] These system calls were developed at the beginning, when no
3330132 //     standard I/O was available. They are to be considered as a
3330133 //     last resort for debugging purposes.
3330134 //
3330135 //---------------------------------------------------------------------
3330136 typedef struct {
3330137     char         path[PATH_MAX];
3330138     int          ret;
3330139     int          errno;
3330140     int          errln;
```

```
    char         errfn[PATH_MAX];
} sysmsg_chdir_t;
//------------------------------------------------------------------------
typedef struct {
    char         path[PATH_MAX];
    mode_t       mode;
    int          ret;
    int          errno;
    int          errln;
    char         errfn[PATH_MAX];
} sysmsg_chmod_t;
//------------------------------------------------------------------------
typedef struct {
    char         path[PATH_MAX];
    uid_t        uid;
    uid_t        gid;
    int          ret;
    int          errno;
    int          errln;
    char         errfn[PATH_MAX];
} sysmsg_chown_t;
//------------------------------------------------------------------------
typedef struct {
    clock_t      ret;
} sysmsg_clock_t;
//------------------------------------------------------------------------
typedef struct {
    int          fdn;
    int          ret;
    int          errno;
    int          errln;
    char         errfn[PATH_MAX];
} sysmsg_close_t;
//------------------------------------------------------------------------
typedef struct {
    int          fdn_old;
    int          ret;
    int          errno;
    int          errln;
    char         errfn[PATH_MAX];
} sysmsg_dup_t;
//------------------------------------------------------------------------
typedef struct {
```

```
3330184     int           fdn_old;
3330185     int           fdn_new;
3330186     int           ret;
3330187     int           errno;
3330188     int           errln;
3330189     char          errfn[PATH_MAX];
3330190 } sysmsg_dup2_t;
3330191 //-----------------------------------------------------------------------
3330192 typedef struct {
3330193     char          path[PATH_MAX];
3330194     int           argc;
3330195     int           envc;
3330196     char          arg_data[ARG_MAX/2];
3330197     char          env_data[ARG_MAX/2];
3330198     uid_t         uid;
3330199     uid_t         euid;
3330200     int           ret;
3330201     int           errno;
3330202     int           errln;
3330203     char          errfn[PATH_MAX];
3330204 } sysmsg_exec_t;
3330205 //-----------------------------------------------------------------------
3330206 typedef struct {
3330207     int           status;
3330208 } sysmsg_exit_t;
3330209 //-----------------------------------------------------------------------
3330210 typedef struct {
3330211     int           fdn;
3330212     mode_t        mode;
3330213     int           ret;
3330214     int           errno;
3330215     int           errln;
3330216     char          errfn[PATH_MAX];
3330217 } sysmsg_fchmod_t;
3330218 //-----------------------------------------------------------------------
3330219 typedef struct {
3330220     int           fdn;
3330221     uid_t         uid;
3330222     uid_t         gid;
3330223     int           ret;
3330224     int           errno;
3330225     int           errln;
3330226     char          errfn[PATH_MAX];
```

```
} sysmsg_fchown_t;
//------------------------------------------------------------------------
typedef struct {
    int          fdn;
    int          cmd;
    int          arg;
    int          ret;
    int          errno;
    int          errln;
    char         errfn[PATH_MAX];
} sysmsg_fcntl_t;
//------------------------------------------------------------------------
typedef struct {
    pid_t        ret;
    int          errno;
    int          errln;
    char         errfn[PATH_MAX];
} sysmsg_fork_t;
//------------------------------------------------------------------------
typedef struct {
    int          fdn;
    struct stat  stat;
    int          ret;
    int          errno;
    int          errln;
    char         errfn[PATH_MAX];
} sysmsg_fstat_t;
//------------------------------------------------------------------------
typedef struct {
    pid_t        pid;
    int          signal;
    int          ret;
    int          errno;
    int          errln;
    char         errfn[PATH_MAX];
} sysmsg_kill_t;
//------------------------------------------------------------------------
typedef struct {
    char         path_old[PATH_MAX];
    char         path_new[PATH_MAX];
    int          ret;
    int          errno;
    int          errln;
```

```
    char         errfn[PATH_MAX];
} sysmsg_link_t;
//-----------------------------------------------------------------------
typedef struct {
    int          fdn;
    off_t        offset;
    int          whence;
    int          ret;
    int          errno;
    int          errln;
    char         errfn[PATH_MAX];
} sysmsg_lseek_t;
//-----------------------------------------------------------------------
typedef struct {
    char         path[PATH_MAX];
    mode_t       mode;
    int          ret;
    int          errno;
    int          errln;
    char         errfn[PATH_MAX];
} sysmsg_mkdir_t;
//-----------------------------------------------------------------------
typedef struct {
    char         path[PATH_MAX];
    mode_t       mode;
    dev_t        device;
    int          ret;
    int          errno;
    int          errln;
    char         errfn[PATH_MAX];
} sysmsg_mknod_t;
//-----------------------------------------------------------------------
typedef struct {
    char         path_dev[PATH_MAX];
    char         path_mnt[PATH_MAX];
    int          options;
    int          ret;
    int          errno;
    int          errln;
    char         errfn[PATH_MAX];
} sysmsg_mount_t;
//-----------------------------------------------------------------------
typedef struct {
```

```
    char         path[PATH_MAX];
    int          flags;
    mode_t       mode;
    int          ret;
    int          errno;
    int          errln;
    char         errfn[PATH_MAX];
} sysmsg_open_t;
//------------------------------------------------------------------------
typedef struct {
    int          fdn;
    char         buffer[BUFSIZ];
    size_t       count;
    int          eof;
    ssize_t      ret;
    int          errno;
    int          errln;
    char         errfn[PATH_MAX];
} sysmsg_read_t;
//------------------------------------------------------------------------
typedef struct {
    uid_t        euid;
    int          ret;
    int          errno;
    int          errln;
    char         errfn[PATH_MAX];
} sysmsg_seteuid_t;
//------------------------------------------------------------------------
typedef struct {
    uid_t        uid;
    uid_t        euid;
    uid_t        suid;
    int          ret;
    int          errno;
    int          errln;
    char         errfn[PATH_MAX];
} sysmsg_setuid_t;
//------------------------------------------------------------------------
typedef struct {
    sighandler_t handler;
    int          signal;
    sighandler_t ret;
    int          errno;
```

```
3330356     int          errln;
3330357     char         errfn[PATH_MAX];
3330358 } sysmsg_signal_t;
3330359 //-------------------------------------------------------------------------
3330360 #define WAKEUP_EVENT_SIGNAL  1    // 1, 2, 4, 8, 16,...
3330361 #define WAKEUP_EVENT_TIMER   2    // so that can be 'OR' combined.
3330362 #define WAKEUP_EVENT_TTY     4    //
3330363 typedef struct {
3330364     int          events;
3330365     int          signal;
3330366     unsigned int seconds;
3330367     time_t       ret;
3330368 } sysmsg_sleep_t;
3330369 //-------------------------------------------------------------------------
3330370 typedef struct {
3330371     char         path[PATH_MAX];
3330372     struct stat  stat;
3330373     int          ret;
3330374     int          errno;
3330375     int          errln;
3330376     char         errfn[PATH_MAX];
3330377 } sysmsg_stat_t;
3330378 //-------------------------------------------------------------------------
3330379 typedef struct {
3330380     time_t       ret;
3330381 } sysmsg_time_t;
3330382 //-------------------------------------------------------------------------
3330383 typedef struct {
3330384     time_t       timer;
3330385     int          ret;
3330386 } sysmsg_stime_t;
3330387 //-------------------------------------------------------------------------
3330388 typedef struct {
3330389     uid_t        uid;         // Read user ID.
3330390     uid_t        euid;        // Effective user ID.
3330391     uid_t        suid;        // Saved user ID.
3330392     pid_t        pid;         // Process ID.
3330393     pid_t        ppid;        // Parent PID.
3330394     pid_t        pgrp;        // Process group.
3330395     mode_t       umask;       // Access permission mask.
3330396     char         path_cwd[PATH_MAX];
3330397 } sysmsg_uarea_t;
3330398 //-------------------------------------------------------------------------
```

```
3330399 typedef struct {
3330400     mode_t       umask;
3330401     mode_t       ret;
3330402 } sysmsg_umask_t;
3330403 //------------------------------------------------------------------------
3330404 typedef struct {
3330405     char         path_mnt[PATH_MAX];
3330406     int          ret;
3330407     int          errno;
3330408     int          errln;
3330409     char         errfn[PATH_MAX];
3330410 } sysmsg_umount_t;
3330411 //------------------------------------------------------------------------
3330412 typedef struct {
3330413     char         path[PATH_MAX];
3330414     int          ret;
3330415     int          errno;
3330416     int          errln;
3330417     char         errfn[PATH_MAX];
3330418 } sysmsg_unlink_t;
3330419 //------------------------------------------------------------------------
3330420 typedef struct {
3330421     int          status;
3330422     pid_t        ret;
3330423     int          errno;
3330424     int          errln;
3330425     char         errfn[PATH_MAX];
3330426 } sysmsg_wait_t;
3330427 //------------------------------------------------------------------------
3330428 typedef struct {
3330429     int          fdn;
3330430     char         buffer[BUFSIZ];
3330431     size_t       count;
3330432     ssize_t      ret;
3330433     int          errno;
3330434     int          errln;
3330435     char         errfn[PATH_MAX];
3330436 } sysmsg_write_t;
3330437 //------------------------------------------------------------------------
3330438 typedef struct {
3330439     char         c;
3330440 } sysmsg_zpchar_t;
3330441 //------------------------------------------------------------------------
```

```
3330442 typedef struct {
3330443     char         string[BUFSIZ];
3330444 } sysmsg_zpstring_t;
3330445 //---------------------------------------------------------------------
3330446 void  heap_clear   (void);
3330447 int   heap_min     (void);
3330448 void  input_line   (char *line, char *prompt, size_t size, int type);
3330449 int   mount        (const char *path_dev, const char *path_mnt,
3330450                     int options);
3330451 int   namep        (const char *name, char *path, size_t size);
3330452 void  process_info (void);
3330453 void  sys          (int syscallnr, void *message, size_t size);
3330454 int   umount       (const char *path_mnt);
3330455 void  z_perror     (const char *string);
3330456 int   z_printf     (const char *restrict format, ...);
3330457 int   z_putchar    (int c);
3330458 int   z_puts       (const char *string);
3330459 int   z_vprintf    (const char *restrict format, va_list arg);
3330460 //int  z_vsprintf   (char *restrict string, const char *restrict format,
3330461 //                   va_list arg);
3330462 //---------------------------------------------------------------------
3330463
3330464 #endif
```

lib/sys/os16/_bp.s

«

Si veda la sezione u0.12.

```
3340001 .global __bp
3340002 .text
3340003 ;---------------------------------------------------------------------
3340004 ; Read the base pointer, as it is before this call.
3340005 ;---------------------------------------------------------------------
3340006 .align 2
3340007 __bp:
3340008     enter #2, #0        ; 1 local variable.
3340009     pushf
3340010     cli
3340011     pusha
3340012     mov   ax, [bp]      ; The previous BP value is saved at *BP.
3340013     mov   -2[bp], ax    ; Save the calculated old SP value.
3340014     popa
```

```
3340015     popf
3340016     mov   ax, -2[bp]    ; AX is the function return value.
3340017     leave
3340018     ret
```

lib/sys/os16/_cs.s

Si veda la sezione u0.12.

```
3350001 .global __cs
3350002 .text
3350003 ;--------------------------------------------------------------------
3350004 ; Read the code segment value.
3350005 ;--------------------------------------------------------------------
3350006 .align 2
3350007 __cs:
3350008     mov ax, cs
3350009     ret
```

lib/sys/os16/_ds.s

Si veda la sezione u0.12.

```
3360001 .global __ds
3360002 .text
3360003 ;--------------------------------------------------------------------
3360004 ; Read the data segment value.
3360005 ;--------------------------------------------------------------------
3360006 .align 2
3360007 __ds:
3360008     mov ax, ds
3360009     ret
```

lib/sys/os16/_es.s

Si veda la sezione u0.12.

```
3370001 .global __es
3370002 .text
3370003 ;--------------------------------------------------------------------
```

```
3370004  ; Read the extra segment value.
3370005  ;--------------------------------------------------------------------
3370006  .align 2
3370007  __es:
3370008      mov ax, es
3370009      ret
```

lib/sys/os16/_seg_d.s

«

Si veda la sezione u0.91.

```
3380001  .global __seg_d
3380002  .text
3380003  ;--------------------------------------------------------------------
3380004  ; Read the data segment value.
3380005  ;--------------------------------------------------------------------
3380006  .align 2
3380007  __seg_d:
3380008      mov ax, ds
3380009      ret
```

lib/sys/os16/_seg_i.s

«

Si veda la sezione u0.91.

```
3390001  .global __seg_i
3390002  .text
3390003  ;--------------------------------------------------------------------
3390004  ; Read the instruction segment value.
3390005  ;--------------------------------------------------------------------
3390006  .align 2
3390007  __seg_i:
3390008      mov ax, cs
3390009      ret
```

lib/sys/os16/_sp.s

«

Si veda la sezione u0.12.

```
3400001  .global __sp
3400002  .text
3400003  ;--------------------------------------------------------------------
3400004  ; Read the stack pointer, as it is before this call.
3400005  ;--------------------------------------------------------------------
3400006  .align 2
3400007  __sp:
3400008      enter #2, #0        ; 1 local variable.
3400009      pushf
3400010      cli
3400011      pusha
3400012      mov   ax, bp        ; The previous SP is equal to BP + 2 + 2.
3400013      add   ax, #4        ;
3400014      mov   -2[bp], ax    ; Save the calculated old SP value.
3400015      popa
3400016      popf
3400017      mov   ax, -2[bp]    ; AX is the function return value.
3400018      leave
3400019      ret
```

lib/sys/os16/_ss.s

«

Si veda la sezione u0.12.

```
3410001  .global __ss
3410002  .text
3410003  ;--------------------------------------------------------------------
3410004  ; Read the stack segment value.
3410005  ;--------------------------------------------------------------------
3410006  .align 2
3410007  __ss:
3410008      mov ax, ss
3410009      ret
```

lib/sys/os16/heap_clear.c

«

Si veda la sezione u0.57.

```
3420001  #include <sys/os16.h>
3420002  //-------------------------------------------------------------------------
3420003  extern uint16_t _end;
3420004  //-------------------------------------------------------------------------
3420005  void heap_clear (void)
3420006  {
3420007      uint16_t *a = &_end;
3420008      uint16_t *z = (void *) (sp () - 2);
3420009      for (; a < z; a++)
3420010        {
3420011          *a = 0xFFFF;
3420012        }
3420013  }
```

lib/sys/os16/heap_min.c

«

Si veda la sezione u0.57.

```
3430001  #include <sys/os16.h>
3430002  //-------------------------------------------------------------------------
3430003  extern uint16_t _end;
3430004  //-------------------------------------------------------------------------
3430005  int heap_min (void)
3430006  {
3430007      uint16_t *a = &_end;
3430008      uint16_t *z = (void *) (sp () - 2);
3430009      int     count;
3430010      for (count = 0; a < z && *a == 0xFFFF; a++, count++);
3430011      return (count * 2);
3430012  }
```

lib/sys/os16/input_line.c

«

Si veda la sezione u0.60.

```
3440001  #include <sys/os16.h>
3440002  #include <string.h>
3440003  #include <stdio.h>
```

```
3440004 //---------------------------------------------------------------------
3440005 void
3440006 input_line (char *line, char *prompt, size_t size, int type)
3440007 {
3440008     int   i;    // Index inside the 'line[]' array.
3440009     int   c;    // Character received from keyboard.
3440010
3440011     if (prompt != NULL || strlen (prompt) > 0)
3440012       {
3440013         printf ("%s ", prompt);
3440014       }
3440015     //
3440016     // Loop for character input. Please note that the loop
3440017     // will exit only through 'break', where the input line
3440018     // will also be correctly terminated with '\0'.
3440019     //
3440020     for (i = 0;; i++)
3440021       {
3440022         c = getchar ();
3440023         //
3440024         // Control codes.
3440025         //
3440026         if (c == EOF)
3440027           {
3440028             line[i] = 0;
3440029             break;
3440030           }
3440031         else if (c == 4)        // [Ctrl D]
3440032           {
3440033             line[i] = 0;
3440034             break;
3440035           }
3440036         else if (c == 10)       // [Enter]
3440037           {
3440038             line[i] = 0;
3440039             break;
3440040           }
3440041         else if (c == 8)        // [Backspace]
3440042           {
3440043             if (i == 0)
3440044               {
3440045                 //
3440046                 // It is already the lowest position, so the video
```

```
3440047                     // cursor is moved forward again, so that the prompt
3440048                     // is not overwritten.
3440049                     // The index is set to -1, so that on the next loop,
3440050                     // it will be again zero.
3440051                     //
3440052                     printf (" ");
3440053                     i = -1;
3440054                   }
3440055                 else
3440056                   {
3440057                     i -= 2;
3440058                   }
3440059                 continue;
3440060               }
3440061             //
3440062             // If 'i' is equal 'size - 1', it is not allowed to continue
3440063             // typing.
3440064             //
3440065             if (i == (size - 1))
3440066               {
3440067                 //
3440068                 // Ignore typing, move back the cursor, delete the character
3440069                 // typed and move back again.
3440070                 //
3440071                 printf ("\b \b");
3440072                 i--;
3440073                 continue;
3440074               }
3440075             //
3440076             // Typing is allowed.
3440077             //
3440078             line[i] = c;
3440079             //
3440080             // Verify if it should be hidden.
3440081             //
3440082             if (type == INPUT_LINE_HIDDEN)
3440083               {
3440084                 printf ("\b ");     // Space: at least you see something.
3440085               }
3440086             else if (type == INPUT_LINE_STARS)
3440087               {
3440088                 printf ("\b*");
3440089               }
```

```
3440090         }
3440091 }
```

lib/sys/os16/mount.c

Si veda la sezione u0.27.

```
3450001 #include <sys/types.h>
3450002 #include <errno.h>
3450003 #include <sys/os16.h>
3450004 #include <stddef.h>
3450005 #include <string.h>
3450006 #include <const.h>
3450007 //---------------------------------------------------------------------
3450008 int
3450009 mount (const char *path_dev, const char *path_mnt, int options)
3450010 {
3450011     sysmsg_mount_t msg;
3450012     //
3450013     strncpy (msg.path_dev, path_dev, PATH_MAX);
3450014     strncpy (msg.path_mnt, path_mnt, PATH_MAX);
3450015     msg.options = options;
3450016     msg.ret     = 0;
3450017     msg.errno   = 0;
3450018     //
3450019     sys (SYS_MOUNT, &msg, (sizeof msg));
3450020     //
3450021     errno = msg.errno;
3450022     errln = msg.errln;
3450023     strncpy (errfn, msg.errfn, PATH_MAX);
3450024     return (msg.ret);
3450025 }
```

lib/sys/os16/namep.c

Si veda la sezione u0.74.

```
3460001 #include <sys/os16.h>
3460002 #include <stdlib.h>
3460003 #include <errno.h>
3460004 #include <unistd.h>
```

```
3460005 //------------------------------------------------------------------------
3460006 int
3460007 namep (const char *name, char *path, size_t size)
3460008 {
3460009     char  command[PATH_MAX];
3460010     char *env_path;
3460011     int   p;            // Index used inside the path environment.
3460012     int   c;            // Index used inside the command string.
3460013     int   status;
3460014     //
3460015     // Check for valid input.
3460016     //
3460017     if (name == NULL || name[0] == 0 || path == NULL || name == path)
3460018       {
3460019         errset (EINVAL);                // Invalid argument.
3460020         return (-1);
3460021       }
3460022     //
3460023     // Check if the original command contains at least a '/'. Otherwise
3460024     // a scan for the environment variable 'PATH' must be done.
3460025     //
3460026     if (strchr (name, '/') == NULL)
3460027       {
3460028         //
3460029         // Ok: no '/' there. Get the environment variable 'PATH'.
3460030         //
3460031         env_path = getenv ("PATH");
3460032         if (env_path == NULL)
3460033           {
3460034             //
3460035             // There is no 'PATH' environment value.
3460036             //
3460037             errset (ENOENT);            // No such file or directory.
3460038             return (-1);
3460039           }
3460040         //
3460041         // Scan paths and try to find a file with that name.
3460042         //
3460043         for (p = 0; env_path[p] != 0;)
3460044           {
3460045             for (c = 0;
3460046                  c < (PATH_MAX - strlen(name) - 2) &&
3460047                  env_path[p] != 0                  &&
```

```
3460048                  env_path[p] != ':';
3460049                  c++, p++)
3460050               {
3460051                 command[c] = env_path[p];
3460052               }
3460053             //
3460054             // If the loop is ended because the command array does not
3460055             // have enough room for the full path, then must return an
3460056             // error.
3460057             //
3460058             if (env_path[p] != ':' && env_path[p] != 0)
3460059               {
3460060                 errset (ENAMETOOLONG);  // Filename too long.
3460061                 return (-1);
3460062               }
3460063             //
3460064             // The command array has enough space. At index `c' must
3460065             // place a zero, to terminate current string.
3460066             //
3460067             command[c] = 0;
3460068             //
3460069             // Add the rest of the path.
3460070             //
3460071             strcat (command, "/");
3460072             strcat (command, name);
3460073             //
3460074             // Verify to have something with that full path name.
3460075             //
3460076             status = access (command, F_OK);
3460077             if (status == 0)
3460078               {
3460079                 //
3460080                 // Verify to have enough room inside the destination
3460081                 // path.
3460082                 //
3460083                 if (strlen (command) >= size)
3460084                   {
3460085                     //
3460086                     // Sorry: too big. There must be room also for
3460087                     // the string termination null character.
3460088                     //
3460089                     errset (ENAMETOOLONG);      // Filename too long.
3460090                     return (-1);
```

```
3460091                     }
3460092                   //
3460093                   // Copy the path and return.
3460094                   //
3460095                   strncpy (path, command, size);
3460096                   return (0);
3460097                 }
3460098               //
3460099               // That path was not good: try again. But before returning
3460100               // to the external loop, must verify if 'p' is to be
3460101               // incremented, after a ':', because the external loop
3460102               // does not touch the index 'p',
3460103               //
3460104               if (env_path[p] == ':')
3460105                 {
3460106                   p++;
3460107                 }
3460108             }
3460109           //
3460110           // At this point, there is no match with the paths.
3460111           //
3460112           errset (ENOENT);                // No such file or directory.
3460113           return (-1);
3460114         }
3460115       //
3460116       // At this point, a path was given and the environment variable
3460117       // 'PATH' was not scanned. Just copy the same path. But must verify
3460118       // that the receiving path has enough room for it.
3460119       //
3460120       if (strlen (name) >= size)
3460121         {
3460122           //
3460123           // Sorry: too big.
3460124           //
3460125           errset (ENAMETOOLONG);  // Filename too long.
3460126           return (-1);
3460127         }
3460128       //
3460129       // Ok: copy and return.
3460130       //
3460131       strncpy (path, name, size);
3460132       return (0);
```

```
3460133 }
```

lib/sys/os16/process_info.c

«

Si veda la sezione u0.79.

```
3470001 #include <sys/os16.h>
3470002 #include <stdio.h>
3470003
3470004 extern uint16_t _edata;
3470005 extern uint16_t _end;
3470006 //---------------------------------------------------------------------
3470007 void
3470008 process_info (void)
3470009 {
3470010     printf ("cs=%04x ds=%04x ss=%04x es=%04x bp=%04x sp=%04x ",
3470011             cs (), ds (), ss (), es (), bp (), sp ());
3470012     printf ("edata=%04x ebss=%04x heap=%04x\n",
3470013             (int) &_edata, (int) &_end, heap_min ());
3470014 }
```

lib/sys/os16/sys.s

«

Si veda la sezione u0.37.

```
3480001 .global _sys
3480002 .text
3480003 ;---------------------------------------------------------------------
3480004 ; Call a system call.
3480005 ;
3480006 ; Please remember that system calls should never be used (called) inside
3480007 ; the kernel code, because system calls cannot be nested for the os16
3480008 ; simple architecture!
3480009 ; If a particular function is necessary inside the kernel, that usually
3480010 ; is made by a system call, an appropriate k_...() function must be
3480011 ; made, to avoid the problem.
3480012 ;
3480013 ;---------------------------------------------------------------------
3480014 .align 2
3480015 _sys:
3480016     int   #0x80
```

```
3480017       ret
```

lib/sys/os16/umount.c

«

Si veda la sezione u0.27.

```
3490001  #include <sys/types.h>
3490002  #include <errno.h>
3490003  #include <sys/os16.h>
3490004  #include <stddef.h>
3490005  #include <string.h>
3490006  //---------------------------------------------------------------------
3490007  int
3490008  umount (const char *path_mnt)
3490009  {
3490010      sysmsg_umount_t msg;
3490011      //
3490012      strncpy (msg.path_mnt, path_mnt, PATH_MAX);
3490013      msg.ret     = 0;
3490014      msg.errno   = 0;
3490015      //
3490016      sys (SYS_UMOUNT, &msg, (sizeof msg));
3490017      //
3490018      errno = msg.errno;
3490019      errln = msg.errln;
3490020      strncpy (errfn, msg.errfn, PATH_MAX);
3490021      return (msg.ret);
3490022  }
```

lib/sys/os16/z_perror.c

«

Si veda la sezione u0.45.

```
3500001  #include <sys/os16.h>
3500002  #include <errno.h>
3500003  #include <stddef.h>
3500004  #include <string.h>
3500005  //---------------------------------------------------------------------
3500006  void
3500007  z_perror (const char *string)
3500008  {
```

```
3500009       //
3500010       // If errno is zero, there is nothing to show.
3500011       //
3500012       if (errno == 0)
3500013         {
3500014           return;
3500015         }
3500016       //
3500017       // Show the string if there is one.
3500018       //
3500019       if (string != NULL && strlen (string) > 0)
3500020         {
3500021           z_printf ("%s: ", string);
3500022         }
3500023       //
3500024       // Show the translated error.
3500025       //
3500026       if (errfn[0] != 0 && errln != 0)
3500027         {
3500028           z_printf ("[%s:%u:%i] %s\n",
3500029                     errfn, errln, errno, strerror (errno));
3500030         }
3500031       else
3500032         {
3500033           z_printf ("[%i] %s\n", errno, strerror (errno));
3500034         }
3500035   }
```

lib/sys/os16/z_printf.c

«

Si veda la sezione u0.45.

```
3510001   #include <sys/os16.h>
3510002   //---------------------------------------------------------------------
3510003   int
3510004   z_printf (char *format, ...)
3510005   {
3510006       va_list ap;
3510007       va_start (ap, format);
3510008       return z_vprintf (format, ap);
3510009   }
```

lib/sys/os16/z_putchar.c

«

Si veda la sezione u0.45.

```
3520001  #include <sys/os16.h>
3520002  //-------------------------------------------------------------------
3520003  int
3520004  z_putchar (int c)
3520005  {
3520006      sysmsg_zpchar_t msg;
3520007      msg.c = c;
3520008      sys (SYS_ZPCHAR, &msg, (sizeof msg));
3520009      return (c);
3520010  }
```

lib/sys/os16/z_puts.c

«

Si veda la sezione u0.45.

```
3530001  #include <sys/os16.h>
3530002  //-------------------------------------------------------------------
3530003  int
3530004  z_puts (char *string)
3530005  {
3530006      unsigned int i;
3530007      for (i = 0; string[i] != 0; string++)
3530008        {
3530009          z_putchar ((int) string[i]);
3530010        }
3530011      z_putchar ((int) '\n');
3530012      return (1);
3530013  }
```

lib/sys/os16/z_vprintf.c

«

Si veda la sezione u0.45.

```
3540001  #include <sys/os16.h>
3540002  //-------------------------------------------------------------------
3540003  int
3540004  z_vprintf (char *format, va_list arg)
3540005  {
```

```
3540006     int                         ret;
3540007     sysmsg_zpstring_t msg;
3540008     msg.string[0] = 0;
3540009     ret = vsprintf (msg.string, format, arg);
3540010     sys (SYS_ZPSTRING, &msg, (sizeof msg));
3540011     return ret;
3540012 }
```

os16: «lib/sys/stat.h»

«

Si veda la sezione u0.2.

```
3550001 #ifndef _SYS_STAT_H
3550002 #define _SYS_STAT_H     1
3550003
3550004 #include <restrict.h>
3550005 #include <const.h>
3550006 #include <sys/types.h>  // dev_t
3550007                         // off_t
3550008                         // blkcnt_t
3550009                         // blksize_t
3550010                         // ino_t
3550011                         // mode_t
3550012                         // nlink_t
3550013                         // uid_t
3550014                         // gid_t
3550015                         // time_t
3550016 //------------------------------------------------------------------------
3550017 // File type.
3550018 //------------------------------------------------------------------------
3550019 #define S_IFMT   0170000        // File type mask.
3550020 //
3550021 #define S_IFBLK  0060000        // Block device file.
3550022 #define S_IFCHR  0020000        // Character device file.
3550023 #define S_IFIFO  0010000        // Pipe (FIFO) file.
3550024 #define S_IFREG  0100000        // Regular file.
3550025 #define S_IFDIR  0040000        // Directory.
3550026 #define S_IFLNK  0120000        // Symbolic link.
3550027 #define S_IFSOCK 0140000        // Unix domain socket.
3550028 //------------------------------------------------------------------------
3550029 // Owner user access permissions.
3550030 //------------------------------------------------------------------------
```

```
3550031 #define S_IRWXU  0000700        // Owner user access permissions mask.
3550032 //
3550033 #define S_IRUSR  0000400        // Owner user read access permission.
3550034 #define S_IWUSR  0000200        // Owner user write access permission.
3550035 #define S_IXUSR  0000100        // Owner user execution or cross perm.
3550036 //------------------------------------------------------------------------
3550037 // Group owner access permissions.
3550038 //------------------------------------------------------------------------
3550039 #define S_IRWXG  0000070        // Owner group access permissions mask.
3550040 //
3550041 #define S_IRGRP  0000040        // Owner group read access permission.
3550042 #define S_IWGRP  0000020        // Owner group write access permission.
3550043 #define S_IXGRP  0000010        // Owner group execution or cross perm.
3550044 //------------------------------------------------------------------------
3550045 // Other users access permissions.
3550046 //------------------------------------------------------------------------
3550047 #define S_IRWXO  0000007        // Other users access permissions mask.
3550048 //
3550049 #define S_IROTH  0000004        // Other users read access permission.
3550050 #define S_IWOTH  0000002        // Other users write access permissions.
3550051 #define S_IXOTH  0000001        // Other users execution or cross perm.
3550052 //------------------------------------------------------------------------
3550053 // S-bit: in this case there is no mask to select all of them.
3550054 //------------------------------------------------------------------------
3550055 #define S_ISUID  0004000        // S-UID.
3550056 #define S_ISGID  0002000        // S-GID.
3550057 #define S_ISVTX  0001000        // Sticky.
3550058 //------------------------------------------------------------------------
3550059 // Macro-instructions to verify the type of file.
3550060 //------------------------------------------------------------------------
3550061 #define S_ISBLK(m)   (((m) & S_IFMT) == S_IFBLK)   // Block device.
3550062 #define S_ISCHR(m)   (((m) & S_IFMT) == S_IFCHR)   // Character device.
3550063 #define S_ISFIFO(m)  (((m) & S_IFMT) == S_IFIFO)   // FIFO.
3550064 #define S_ISREG(m)   (((m) & S_IFMT) == S_IFREG)   // Regular file.
3550065 #define S_ISDIR(m)   (((m) & S_IFMT) == S_IFDIR)   // Directory.
3550066 #define S_ISLNK(m)   (((m) & S_IFMT) == S_IFLNK)   // Symbolic link.
3550067 #define S_ISSOCK(m)  (((m) & S_IFMT) == S_IFSOCK)  // Socket.
3550068 //------------------------------------------------------------------------
3550069 // Structure 'stat'.
3550070 //------------------------------------------------------------------------
3550071 struct stat {
3550072     dev_t     st_dev;       // Device containing the file.
3550073     ino_t     st_ino;       // File serial number (inode number).
```

```
3550074      mode_t    st_mode;      // File type and permissions.
3550075      nlink_t   st_nlink;     // Links to the file.
3550076      uid_t     st_uid;       // Owner user id.
3550077      gid_t     st_gid;       // Owner group id.
3550078      dev_t     st_rdev;      // Device number if it is a device file.
3550079      off_t     st_size;      // File size.
3550080      time_t    st_atime;     // Last access time.
3550081      time_t    st_mtime;     // Last modification time.
3550082      time_t    st_ctime;     // Last inode modification.
3550083      blksize_t st_blksize;   // Block size for I/O operations.
3550084      blkcnt_t  st_blocks;    // File size / block size.
3550085  };
3550086  //-----------------------------------------------------------------------
3550087  // Function prototypes.
3550088  //-----------------------------------------------------------------------
3550089  int    chmod  (const char *path, mode_t mode);
3550090  int    fchmod (int fdn, mode_t mode);
3550091  int    fstat  (int fdn, struct stat *buffer);
3550092  int    lstat  (const char *restrict path, struct stat *restrict buffer);
3550093  int    mkdir  (const char *path, mode_t mode);
3550094  int    mkfifo (const char *path, mode_t mode);
3550095  int    mknod  (const char *path, mode_t mode, dev_t dev);
3550096  int    stat   (const char *restrict path, struct stat *restrict buffer);
3550097  mode_t umask  (mode_t mask);
3550098
3550099  #endif // _SYS_STAT_H
```

lib/sys/stat/chmod.c

«

Si veda la sezione u0.4.

```
3560001  #include <sys/stat.h>
3560002  #include <string.h>
3560003  #include <const.h>
3560004  #include <sys/os16.h>
3560005  #include <errno.h>
3560006  #include <limits.h>
3560007  //-----------------------------------------------------------------------
3560008  int
3560009  chmod (const char *path, mode_t mode)
3560010  {
3560011      sysmsg_chmod_t msg;
```

```
3560012     //
3560013     strncpy (msg.path, path, PATH_MAX);
3560014     msg.mode = mode;
3560015     //
3560016     sys (SYS_CHMOD, &msg, (sizeof msg));
3560017     //
3560018     errno = msg.errno;
3560019     errln = msg.errln;
3560020     strncpy (errfn, msg.errfn, PATH_MAX);
3560021     return (msg.ret);
3560022 }
```

lib/sys/stat/fchmod.c

«

Si veda la sezione u0.4.

```
3570001 #include <sys/stat.h>
3570002 #include <string.h>
3570003 #include <const.h>
3570004 #include <sys/os16.h>
3570005 #include <errno.h>
3570006 #include <limits.h>
3570007 //------------------------------------------------------------------------
3570008 int
3570009 fchmod (int fdn, mode_t mode)
3570010 {
3570011     sysmsg_fchmod_t msg;
3570012     //
3570013     msg.fdn  = fdn;
3570014     msg.mode = mode;
3570015     //
3570016     sys (SYS_FCHMOD, &msg, (sizeof msg));
3570017     //
3570018     errno = msg.errno;
3570019     errln = msg.errln;
3570020     strncpy (errfn, msg.errfn, PATH_MAX);
3570021     return (msg.ret);
3570022 }
```

«

Si veda la sezione u0.36.

```
3580001 #include <unistd.h>
3580002 #include <errno.h>
3580003 #include <sys/os16.h>
3580004 #include <string.h>
3580005 //-------------------------------------------------------------------
3580006 int
3580007 fstat (int fdn, struct stat *buffer)
3580008 {
3580009     sysmsg_fstat_t msg;
3580010     //
3580011     msg.fdn              = fdn;
3580012     msg.stat.st_dev      = buffer->st_dev;
3580013     msg.stat.st_ino      = buffer->st_ino;
3580014     msg.stat.st_mode     = buffer->st_mode;
3580015     msg.stat.st_nlink    = buffer->st_nlink;
3580016     msg.stat.st_uid      = buffer->st_uid;
3580017     msg.stat.st_gid      = buffer->st_gid;
3580018     msg.stat.st_rdev     = buffer->st_rdev;
3580019     msg.stat.st_size     = buffer->st_size;
3580020     msg.stat.st_atime    = buffer->st_atime;
3580021     msg.stat.st_mtime    = buffer->st_mtime;
3580022     msg.stat.st_ctime    = buffer->st_ctime;
3580023     msg.stat.st_blksize  = buffer->st_blksize;
3580024     msg.stat.st_blocks   = buffer->st_blocks;
3580025     //
3580026     sys (SYS_FSTAT, &msg, (sizeof msg));
3580027     //
3580028     buffer->st_dev       = msg.stat.st_dev;
3580029     buffer->st_ino       = msg.stat.st_ino;
3580030     buffer->st_mode      = msg.stat.st_mode;
3580031     buffer->st_nlink     = msg.stat.st_nlink;
3580032     buffer->st_uid       = msg.stat.st_uid;
3580033     buffer->st_gid       = msg.stat.st_gid;
3580034     buffer->st_rdev      = msg.stat.st_rdev;
3580035     buffer->st_size      = msg.stat.st_size;
3580036     buffer->st_atime     = msg.stat.st_atime;
3580037     buffer->st_mtime     = msg.stat.st_mtime;
3580038     buffer->st_ctime     = msg.stat.st_ctime;
3580039     buffer->st_blksize   = msg.stat.st_blksize;
3580040     buffer->st_blocks    = msg.stat.st_blocks;
```

```
3580041     //
3580042     errno = msg.errno;
3580043     errln = msg.errln;
3580044     strncpy (errfn, msg.errfn, PATH_MAX);
3580045     return (msg.ret);
3580046 }
```

lib/sys/stat/mkdir.c

«

Si veda la sezione u0.25.

```
3590001 #include <sys/stat.h>
3590002 #include <string.h>
3590003 #include <const.h>
3590004 #include <sys/os16.h>
3590005 #include <errno.h>
3590006 #include <limits.h>
3590007 //------------------------------------------------------------------------
3590008 int
3590009 mkdir (const char *path, mode_t mode)
3590010 {
3590011     sysmsg_mkdir_t msg;
3590012     //
3590013     strncpy (msg.path, path, PATH_MAX);
3590014     msg.mode = mode;
3590015     //
3590016     sys (SYS_MKDIR, &msg, (sizeof msg));
3590017     //
3590018     errno = msg.errno;
3590019     errln = msg.errln;
3590020     strncpy (errfn, msg.errfn, PATH_MAX);
3590021     return (msg.ret);
3590022 }
```

lib/sys/stat/mknod.c

«

Si veda la sezione u0.26.

```
3600001 #include <unistd.h>
3600002 #include <errno.h>
3600003 #include <sys/os16.h>
```

```
3600004 #include <string.h>
3600005 //-------------------------------------------------------------------------
3600006 int
3600007 mknod (const char *path, mode_t mode, dev_t device)
3600008 {
3600009     sysmsg_mknod_t msg;
3600010     //
3600011     strncpy (msg.path, path, PATH_MAX);
3600012     msg.mode   = mode;
3600013     msg.device = device;
3600014     //
3600015     sys (SYS_MKNOD, &msg, (sizeof msg));
3600016     //
3600017     errno = msg.errno;
3600018     errln = msg.errln;
3600019     strncpy (errfn, msg.errfn, PATH_MAX);
3600020     return (msg.ret);
3600021 }
```

lib/sys/stat/stat.c

«

Si veda la sezione u0.36.

```
3610001 #include <unistd.h>
3610002 #include <errno.h>
3610003 #include <sys/os16.h>
3610004 #include <string.h>
3610005 //-------------------------------------------------------------------------
3610006 int
3610007 stat (const char *path, struct stat *buffer)
3610008 {
3610009     sysmsg_stat_t msg;
3610010     //
3610011     strncpy (msg.path, path, PATH_MAX);
3610012     //
3610013     msg.stat.st_dev     = buffer->st_dev;
3610014     msg.stat.st_ino     = buffer->st_ino;
3610015     msg.stat.st_mode    = buffer->st_mode;
3610016     msg.stat.st_nlink   = buffer->st_nlink;
3610017     msg.stat.st_uid     = buffer->st_uid;
3610018     msg.stat.st_gid     = buffer->st_gid;
3610019     msg.stat.st_rdev    = buffer->st_rdev;
```

```
3610020     msg.stat.st_size    = buffer->st_size;
3610021     msg.stat.st_atime   = buffer->st_atime;
3610022     msg.stat.st_mtime   = buffer->st_mtime;
3610023     msg.stat.st_ctime   = buffer->st_ctime;
3610024     msg.stat.st_blksize = buffer->st_blksize;
3610025     msg.stat.st_blocks  = buffer->st_blocks;
3610026     //
3610027     sys (SYS_STAT, &msg, (sizeof msg));
3610028     //
3610029     buffer->st_dev      = msg.stat.st_dev;
3610030     buffer->st_ino      = msg.stat.st_ino;
3610031     buffer->st_mode     = msg.stat.st_mode;
3610032     buffer->st_nlink    = msg.stat.st_nlink;
3610033     buffer->st_uid      = msg.stat.st_uid;
3610034     buffer->st_gid      = msg.stat.st_gid;
3610035     buffer->st_rdev     = msg.stat.st_rdev;
3610036     buffer->st_size     = msg.stat.st_size;
3610037     buffer->st_atime    = msg.stat.st_atime;
3610038     buffer->st_mtime    = msg.stat.st_mtime;
3610039     buffer->st_ctime    = msg.stat.st_ctime;
3610040     buffer->st_blksize  = msg.stat.st_blksize;
3610041     buffer->st_blocks   = msg.stat.st_blocks;
3610042     //
3610043     errno = msg.errno;
3610044     errln = msg.errln;
3610045     strncpy (errfn, msg.errfn, PATH_MAX);
3610046     return (msg.ret);
3610047 }
```

lib/sys/stat/umask.c

«

Si veda la sezione u0.40.

```
3620001 #include <sys/stat.h>
3620002 #include <string.h>
3620003 #include <const.h>
3620004 #include <sys/os16.h>
3620005 #include <errno.h>
3620006 #include <limits.h>
3620007 //------------------------------------------------------------------------
3620008 mode_t
3620009 umask (mode_t mask)
```

```
3620010 {
3620011     sysmsg_umask_t msg;
3620012     msg.umask = mask;
3620013     sys (SYS_UMASK, &msg, (sizeof msg));
3620014     return (msg.ret);
3620015 }
```

os16: «lib/sys/types.h»

«

Si veda la sezione u0.2.

```
3630001 #ifndef _SYS_TYPES_H
3630002 #define _SYS_TYPES_H    1
3630003 //-------------------------------------------------------------------------
3630004
3630005 #include <clock_t.h>
3630006 #include <time_t.h>
3630007 #include <size_t.h>
3630008 #include <stdint.h>
3630009 //-------------------------------------------------------------------------
3630010 typedef     long int blkcnt_t;
3630011 typedef     long int blksize_t;
3630012 typedef     uint16_t dev_t;     // Traditional device size.
3630013 typedef unsigned int id_t;
3630014 typedef unsigned int gid_t;
3630015 typedef unsigned int uid_t;
3630016 typedef     uint16_t ino_t;     // Minix 1 file system inode size.
3630017 typedef     uint16_t mode_t;    // Minix 1 file system mode size.
3630018 typedef unsigned int nlink_t;
3630019 typedef     long int off_t;
3630020 typedef          int pid_t;
3630021 typedef unsigned int pthread_t;
3630022 typedef     long int ssize_t;
3630023 //-------------------------------------------------------------------------
3630024 // Common extentions.
3630025 //
3630026 dev_t makedev (int major, int minor);
3630027 int   major   (dev_t device);
3630028 int   minor   (dev_t device);
3630029 //-------------------------------------------------------------------------
3630030
3630031 #endif
```

lib/sys/types/major.c

«

Si veda la sezione u0.65.

```
3640001  #include <sys/types.h>
3640002  //------------------------------------------------------------------------
3640003  int
3640004  major (dev_t device)
3640005  {
3640006      return ((int) (device / 256));
3640007  }
```

lib/sys/types/makedev.c

«

Si veda la sezione u0.65.

```
3650001  #include <sys/types.h>
3650002  //------------------------------------------------------------------------
3650003  dev_t
3650004  makedev (int major, int minor)
3650005  {
3650006      return ((dev_t) (major * 256 + minor));
3650007  }
```

lib/sys/types/minor.c

«

Si veda la sezione u0.65.

```
3660001  #include <sys/types.h>
3660002  //------------------------------------------------------------------------
3660003  int
3660004  minor (dev_t device)
3660005  {
3660006      return ((dev_t) (device & 0x00FF));
3660007  }
```

os16: «lib/sys/wait.h»

«

Si veda la sezione u0.2.

```
3670001  #ifndef _SYS_WAIT_H
3670002  #define _SYS_WAIT_H      1
3670003
3670004  #include <sys/types.h>
3670005
3670006  //-------------------------------------------------------------------------
3670007  pid_t wait    (int *status);
3670008  //-------------------------------------------------------------------------
3670009
3670010  #endif
```

lib/sys/wait/wait.c

«

Si veda la sezione u0.43.

```
3680001  #include <sys/types.h>
3680002  #include <errno.h>
3680003  #include <sys/os16.h>
3680004  #include <stddef.h>
3680005  #include <string.h>
3680006  //-------------------------------------------------------------------------
3680007  pid_t
3680008  wait (int *status)
3680009  {
3680010      sysmsg_wait_t msg;
3680011      msg.ret    = 0;
3680012      msg.errno  = 0;
3680013      msg.status = 0;
3680014      while (msg.ret == 0)
3680015        {
3680016          //
3680017          // Loop as long as there are children, an none is dead.
3680018          //
3680019          sys (SYS_WAIT, &msg, (sizeof msg));
3680020        }
3680021      errno = msg.errno;
3680022      errln = msg.errln;
3680023      strncpy (errfn, msg.errfn, PATH_MAX);
3680024      //
```

```
3680025     if (status != NULL)
3680026       {
3680027         //
3680028         // Only the low eight bits are returned.
3680029         //
3680030         *status = (msg.status & 0x00FF);
3680031       }
3680032     return (msg.ret);
3680033 }
```

os16: «lib/time.h»

«

Si veda la sezione u0.2.

```
3690001 #ifndef _TIME_H
3690002 #define _TIME_H       1
3690003 //-------------------------------------------------------------------------
3690004
3690005 #include <const.h>
3690006 #include <restrict.h>
3690007 #include <size_t.h>
3690008 #include <time_t.h>
3690009 #include <clock_t.h>
3690010 #include <NULL.h>
3690011 #include <stdint.h>
3690012 //-------------------------------------------------------------------------
3690013 #define CLOCKS_PER_SEC  18  // Should be 18.22 Hz, but it is a 'int'.
3690014 //-------------------------------------------------------------------------
3690015 struct tm {int tm_sec;  int tm_min;  int tm_hour;
3690016            int tm_mday; int tm_mon;  int tm_year;
3690017            int tm_wday; int tm_yday; int tm_isdst;};
3690018 //-------------------------------------------------------------------------
3690019 clock_t     clock     (void);
3690020 time_t      time      (time_t *timer);
3690021 int         stime     (time_t *timer);
3690022 double      difftime  (time_t time1, time_t time0);
3690023 time_t      mktime    (struct tm *timeptr);
3690024 struct tm *gmtime     (const time_t *timer);
3690025 struct tm *localtime  (const time_t *timer);
3690026 char       *asctime   (const struct tm *timeptr);
3690027 char       *ctime     (const time_t *timer);
3690028 size_t      strftime  (char * restrict s, size_t maxsize,
```

```
3690029                         const char * restrict format,
3690030                         const struct tm * restrict timeptr);
3690031 //---------------------------------------------------------------------
3690032 #define difftime(t1,t0) ((double)((t1)-(t0)))
3690033 #define ctime(t)        (asctime (localtime (t)))
3690034 #define localtime(t)    (gmtime (t))
3690035 //---------------------------------------------------------------------
3690036
3690037 #endif
```

lib/time/asctime.c

«

Si veda la sezione u0.13.

```
3700001 #include <time.h>
3700002 #include <string.h>
3700003 #include <stdio.h>
3700004
3700005 //---------------------------------------------------------------------
3700006 char *
3700007 asctime (const struct tm *timeptr)
3700008 {
3700009     static char time_string[25]; // 'Sun Jan 30 24:00:00 2111'
3700010     //
3700011     // Check argument.
3700012     //
3700013     if (timeptr == NULL)
3700014       {
3700015         return (NULL);
3700016       }
3700017     //
3700018     // Set week day.
3700019     //
3700020     switch (timeptr->tm_wday)
3700021       {
3700022         case 0:
3700023             strcpy (&time_string[0], "Sun");
3700024             break;
3700025         case 1:
3700026             strcpy (&time_string[0], "Mon");
3700027             break;
3700028         case 2:
```

```
3700029             strcpy (&time_string[0], "Tue");
3700030             break;
3700031         case 3:
3700032             strcpy (&time_string[0], "Wed");
3700033             break;
3700034         case 4:
3700035             strcpy (&time_string[0], "Thu");
3700036             break;
3700037         case 5:
3700038             strcpy (&time_string[0], "Fri");
3700039             break;
3700040         case 6:
3700041             strcpy (&time_string[0], "Sat");
3700042             break;
3700043         default:
3700044             strcpy (&time_string[0], "Err");
3700045       }
3700046     //
3700047     // Set month.
3700048     //
3700049     switch (timeptr->tm_mon)
3700050       {
3700051         case 1:
3700052             strcpy (&time_string[3], " Jan");
3700053             break;
3700054         case 2:
3700055             strcpy (&time_string[3], " Feb");
3700056             break;
3700057         case 3:
3700058             strcpy (&time_string[3], " Mar");
3700059             break;
3700060         case 4:
3700061             strcpy (&time_string[3], " Apr");
3700062             break;
3700063         case 5:
3700064             strcpy (&time_string[3], " May");
3700065             break;
3700066         case 6:
3700067             strcpy (&time_string[3], " Jun");
3700068             break;
3700069         case 7:
3700070             strcpy (&time_string[3], " Jul");
3700071             break;
```

```
3700072             case 8:
3700073                 strcpy (&time_string[3], " Aug");
3700074                 break;
3700075             case 9:
3700076                 strcpy (&time_string[3], " Sep");
3700077                 break;
3700078             case 10:
3700079                 strcpy (&time_string[3], " Oct");
3700080                 break;
3700081             case 11:
3700082                 strcpy (&time_string[3], " Nov");
3700083                 break;
3700084             case 12:
3700085                 strcpy (&time_string[3], " Dec");
3700086                 break;
3700087             default:
3700088                 strcpy (&time_string[3], " Err");
3700089           }
3700090         //
3700091         // Set day of month, hour, minute, second and year.
3700092         //
3700093         sprintf (&time_string[7], " %2i %2i:%2i:%2i %4i",
3700094                  timeptr->tm_mday,  timeptr->tm_hour,  timeptr->tm_min,
3700095                  timeptr->tm_sec,  timeptr->tm_year);
3700096         //
3700097         //
3700098         //
3700099         return (&time_string[0]);
3700100     }
```

lib/time/clock.c

«

Si veda la sezione u0.6.

```
3710001 #include <time.h>
3710002 #include <sys/os16.h>
3710003 //---------------------------------------------------------------------
3710004 clock_t
3710005 clock (void)
3710006 {
3710007     sysmsg_clock_t msg;
3710008     msg.ret = 0;
```

```
3710009      sys (SYS_CLOCK, &msg, (sizeof msg));
3710010      return (msg.ret);
3710011  }
3710012
```

lib/time/gmtime.c

«

Si veda la sezione u0.13.

```
3720001  #include <time.h>
3720002  //-------------------------------------------------------------------------
3720003  static int leap_year (int year);
3720004  //-------------------------------------------------------------------------
3720005  struct tm *
3720006  gmtime (const time_t *timer)
3720007  {
3720008      static struct tm  tms;
3720009      int               loop;
3720010      unsigned int      remainder;
3720011      unsigned int      days;
3720012      //
3720013      // Check argument.
3720014      //
3720015      if (timer == NULL)
3720016        {
3720017          return (NULL);
3720018        }
3720019      //
3720020      // Days since epoch. There are 86400 seconds per day.
3720021      // At the moment, the field 'tm_yday' will contain
3720022      // all days since epoch.
3720023      //
3720024      days      = *timer / 86400L;
3720025      remainder = *timer % 86400L;
3720026      //
3720027      // Minutes, after full days.
3720028      //
3720029      tms.tm_min = remainder / 60U;
3720030      //
3720031      // Seconds, after full minutes.
3720032      //
3720033      tms.tm_sec = remainder % 60U;
```

```
3720034       //
3720035       // Hours, after full days.
3720036       //
3720037       tms.tm_hour = tms.tm_min / 60;
3720038       //
3720039       // Minutes, after full hours.
3720040       //
3720041       tms.tm_min = tms.tm_min % 60;
3720042       //
3720043       // Find the week day. Must remove some days to align the
3720044       // calculation. So: the week days of the first week of 1970
3720045       // are not valid! After 1970-01-04 calculations are right.
3720046       //
3720047       tms.tm_wday = (days - 3) % 7;
3720048       //
3720049       // Find the year: the field 'tm_yday' will be reduced to the days
3720050       // of current year.
3720051       //
3720052       for (tms.tm_year = 1970; days > 0; tms.tm_year++)
3720053         {
3720054           if (leap_year (tms.tm_year))
3720055             {
3720056               if (days >= 366)
3720057                 {
3720058                   days -= 366;
3720059                   continue;
3720060                 }
3720061               else
3720062                 {
3720063                   break;
3720064                 }
3720065             }
3720066           else
3720067             {
3720068               if (days >= 365)
3720069                 {
3720070                   days -= 365;
3720071                   continue;
3720072                 }
3720073               else
3720074                 {
3720075                   break;
3720076                 }
```

```
            }
        }
      //
      // Day of the year.
      //
      tms.tm_yday = days + 1;
      //
      // Find the month.
      //
      tms.tm_mday = days + 1;
      //
      for (tms.tm_mon = 0, loop = 1; tms.tm_mon <= 12 && loop;)
        {
          tms.tm_mon++;
          //
          switch (tms.tm_mon)
            {
              case 1:
              case 3:
              case 5:
              case 7:
              case 8:
              case 10:
              case 12:
                  if (tms.tm_mday >= 31)
                    {
                      tms.tm_mday -= 31;
                    }
                  else
                    {
                      loop = 0;
                    }
                  break;
              case 4:
              case 6:
              case 9:
              case 11:
                  if (tms.tm_mday >= 30)
                    {
                      tms.tm_mday -= 30;
                    }
                  else
                    {
```

```
                    loop = 0;
                  }
                break;
            case 2:
                if (leap_year (tms.tm_year))
                  {
                    if (tms.tm_mday >= 29)
                      {
                        tms.tm_mday -= 29;
                      }
                    else
                      {
                        loop = 0;
                      }
                  }
                else
                  {
                    if (tms.tm_mday >= 28)
                      {
                        tms.tm_mday -= 28;
                      }
                    else
                      {
                        loop = 0;
                      }
                  }
                break;
            }
        }
    //
    // No check for day light saving time.
    //
    tms.tm_isdst = 0;
    //
    // Return.
    //
    return (&tms);
}
//---------------------------------------------------------------------
static int
leap_year (int year)
{
    if ((year % 4) == 0)
```

```
3720163         {
3720164           if ((year % 100) == 0)
3720165             {
3720166               if ((year % 400) == 0)
3720167                 {
3720168                   return (1);
3720169                 }
3720170               else
3720171                 {
3720172                   return (0);
3720173                 }
3720174             }
3720175           else
3720176             {
3720177               return (1);
3720178             }
3720179         }
3720180       else
3720181         {
3720182           return (0);
3720183         }
3720184   }
```

lib/time/mktime.c

«

Si veda la sezione u0.13.

```
3730001   #include <time.h>
3730002   #include <string.h>
3730003   #include <stdio.h>
3730004   //-------------------------------------------------------------------
3730005   static int leap_year (int year);
3730006   //-------------------------------------------------------------------
3730007   time_t
3730008   mktime (const struct tm *timeptr)
3730009   {
3730010       time_t timer_total;
3730011       time_t timer_aux;
3730012       int    days;
3730013       int    month;
3730014       int    year;
3730015       //
```

```
3730016     // From seconds to days.
3730017     //
3730018     timer_total  = timeptr->tm_sec;
3730019     //
3730020     timer_aux    = timeptr->tm_min;
3730021     timer_aux   *= 60;
3730022     timer_total += timer_aux;
3730023     //
3730024     timer_aux    = timeptr->tm_hour;
3730025     timer_aux   *= (60 * 60);
3730026     timer_total += timer_aux;
3730027     //
3730028     timer_aux    = timeptr->tm_mday;
3730029     timer_aux   *= 24;
3730030     timer_aux   *= (60 * 60);
3730031     timer_total += timer_aux;
3730032     //
3730033     // Month: add the days of months.
3730034     // Will scan the months, from the first, but before the
3730035     // months of the value inside field 'tm_mon'.
3730036     //
3730037     for (month = 1, days = 0; month < timeptr->tm_mon; month++)
3730038       {
3730039         switch (month)
3730040           {
3730041             case 1:
3730042             case 3:
3730043             case 5:
3730044             case 7:
3730045             case 8:
3730046             case 10:
3730047                 //
3730048                 // There is no December, because the scan can go up to
3730049                 // the month before the value inside field 'tm_mon'.
3730050                 //
3730051                 days += 31;
3730052                 break;
3730053             case 4:
3730054             case 6:
3730055             case 9:
3730056             case 11:
3730057                 days += 30;
3730058                 break;
```

```
3730059             case 2:
3730060                 if (leap_year (timeptr->tm_year))
3730061                   {
3730062                     days += 29;
3730063                   }
3730064                 else
3730065                   {
3730066                     days += 28;
3730067                   }
3730068                 break;
3730069           }
3730070       }
3730071     //
3730072     timer_aux    = days;
3730073     timer_aux   *= 24;
3730074     timer_aux   *= (60 * 60);
3730075     timer_total += timer_aux;
3730076     //
3730077     // Year. The work is similar to the one of months: days of
3730078     // years are counted, up to the year before the one reported
3730079     // by the field 'tm_year'.
3730080     //
3730081     for (year = 1970, days = 0; year < timeptr->tm_year; year++)
3730082       {
3730083         if (leap_year (year))
3730084           {
3730085             days += 366;
3730086           }
3730087         else
3730088           {
3730089             days += 365;
3730090           }
3730091       }
3730092     //
3730093     // After all, must subtract a day from the total.
3730094     //
3730095     days--;
3730096     //
3730097     timer_aux    = days;
3730098     timer_aux   *= 24;
3730099     timer_aux   *= (60 * 60);
3730100     timer_total += timer_aux;
3730101     //
```

```
3730102     // That's all.
3730103     //
3730104     return (timer_total);
3730105 }
3730106 //---------------------------------------------------------------------
3730107 int
3730108 leap_year (int year)
3730109 {
3730110     if ((year % 4) == 0)
3730111       {
3730112         if ((year % 100) == 0)
3730113           {
3730114             if ((year % 400) == 0)
3730115               {
3730116                 return (1);
3730117               }
3730118             else
3730119               {
3730120                 return (0);
3730121               }
3730122           }
3730123         else
3730124           {
3730125             return (1);
3730126           }
3730127       }
3730128     else
3730129       {
3730130         return (0);
3730131       }
3730132 }
```

lib/time/stime.c

«

Si veda la sezione u0.39.

```
3740001 #include <time.h>
3740002 #include <sys/os16.h>
3740003 //---------------------------------------------------------------------
3740004 int
3740005 stime (time_t *timer)
3740006 {
```

```
3740007       sysmsg_stime_t msg;
3740008       msg.timer = *timer;
3740009       msg.ret = 0;
3740010       sys (SYS_STIME, &msg, (sizeof msg));
3740011       return (msg.ret);
3740012   }
```

lib/time/time.c

«

Si veda la sezione u0.39.

```
3750001   #include <time.h>
3750002   #include <sys/os16.h>
3750003   //------------------------------------------------------------------------
3750004   time_t
3750005   time (time_t *timer)
3750006   {
3750007       sysmsg_time_t msg;
3750008       msg.ret = ((time_t) 0);
3750009       sys (SYS_TIME, &msg, (sizeof msg));
3750010       if (timer != NULL)
3750011         {
3750012           *timer = msg.ret;
3750013         }
3750014       return (msg.ret);
3750015   }
```

os16: «lib/unistd.h»

«

Si veda la sezione u0.2.

```
3760001   #ifndef _UNISTD_H
3760002   #define _UNISTD_H       1
3760003
3760004   #include <const.h>
3760005   #include <sys/stat.h>
3760006   #include <sys/os16.h>
3760007   #include <sys/types.h>  // size_t, ssize_t, uid_t, gid_t, off_t, pid_t
3760008   #include <inttypes.h>   // intptr_t
3760009   #include <SEEK.h>       // SEEK_CUR, SEEK_SET, SEEK_END
3760010   //------------------------------------------------------------------------
```

```
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          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, const char *arg, ...);
int          execle      (const char *path, const char *arg, ...);
int          execlp      (const char *path, const 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);
uid_t        geteuid     (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);
```

```
3760054  int           link        (const char *path_old, const char *path_new);
3760055  off_t         lseek       (int fdn, off_t offset, int whence);
3760056  #define       nice(n)     (0)
3760057  ssize_t       read        (int fdn, void *buffer, size_t count);
3760058  #define       readlink(p,b,s) ((ssize_t) -1)
3760059  int           rmdir       (const char *path);
3760060  int           seteuid     (uid_t uid);
3760061  int           setpgrp     (void);
3760062  int           setuid      (uid_t uid);
3760063  unsigned int  sleep       (unsigned int s);
3760064  #define       sync()      /**/
3760065  char         *ttyname     (int fdn);
3760066  int           unlink      (const char *path);
3760067  ssize_t       write       (int fdn, const void *buffer, size_t count);
3760068
3760069  #endif
```

lib/unistd/_exit.c

«

Si veda la sezione u0.2.

```
3770001  #include <unistd.h>
3770002  #include <sys/os16.h>
3770003  //------------------------------------------------------------------------
3770004  void
3770005  _exit (int status)
3770006  {
3770007      sysmsg_exit_t msg;
3770008      //
3770009      // Only the low eight bit are returned.
3770010      //
3770011      msg.status = (status & 0xFF);
3770012      //
3770013      //
3770014      //
3770015      sys (SYS_EXIT, &msg, (sizeof msg));
3770016      //
3770017      // Should not return from system call, but if it does, loop
3770018      // forever:
3770019      //
3770020      while (1);
3770021  }
```

«

Si veda la sezione u0.1.

```
3780001 #include <unistd.h>
3780002 #include <sys/stat.h>
3780003 #include <errno.h>
3780004 //---------------------------------------------------------------------
3780005 int
3780006 access (const char *path, int mode)
3780007 {
3780008     struct stat st;
3780009     int         status;
3780010     uid_t       euid;
3780011     //
3780012     status = stat (path, &st);
3780013     if (status != 0)
3780014       {
3780015         return (-1);
3780016       }
3780017     //
3780018     // File exists?
3780019     //
3780020     if (mode == F_OK)
3780021       {
3780022         return (0);
3780023       }
3780024     //
3780025     // Some access permissions are requested: get effective user id.
3780026     //
3780027     euid = geteuid ();
3780028     //
3780029     // Check owner access permissions.
3780030     //
3780031     if (st.st_uid == euid && ((st.st_mode & S_IRWXU) == (mode << 6)))
3780032       {
3780033         return (0);
3780034       }
3780035     //
3780036     // Check others access permissions.
3780037     //
3780038     if ((st.st_mode & S_IRWXO) == (mode))
3780039       {
3780040         return (0);
```

```
3780041         }
3780042     //
3780043     // Otherwise there are no access permissions.
3780044     //
3780045     errset (EACCES);                    // Permission denied.
3780046     return (-1);
3780047 }
```

lib/unistd/chdir.c

«

Si veda la sezione u0.3.

```
3790001 #include <unistd.h>
3790002 #include <string.h>
3790003 #include <const.h>
3790004 #include <sys/os16.h>
3790005 #include <errno.h>
3790006 #include <limits.h>
3790007 //-----------------------------------------------------------------------
3790008 int
3790009 chdir (const char *path)
3790010 {
3790011     sysmsg_chdir_t msg;
3790012     //
3790013     msg.ret    = 0;
3790014     msg.errno  = 0;
3790015     //
3790016     strncpy (msg.path, path, PATH_MAX);
3790017     //
3790018     sys (SYS_CHDIR, &msg, (sizeof msg));
3790019     //
3790020     errno = msg.errno;
3790021     errln = msg.errln;
3790022     strncpy (errfn, msg.errfn, PATH_MAX);
3790023     return (msg.ret);
3790024 }
```

lib/unistd/chown.c

«

Si veda la sezione u0.5.

```
3800001  #include <unistd.h>
3800002  #include <string.h>
3800003  #include <const.h>
3800004  #include <sys/os16.h>
3800005  #include <errno.h>
3800006  #include <limits.h>
3800007  //------------------------------------------------------------------------
3800008  int
3800009  chown (const char *path, uid_t uid, gid_t gid)
3800010  {
3800011      sysmsg_chown_t msg;
3800012      //
3800013      strncpy (msg.path, path, PATH_MAX);
3800014      msg.uid = uid;
3800015      msg.gid = gid;
3800016      //
3800017      sys (SYS_CHOWN, &msg, (sizeof msg));
3800018      //
3800019      errno = msg.errno;
3800020      errln = msg.errln;
3800021      strncpy (errfn, msg.errfn, PATH_MAX);
3800022      return (msg.ret);
3800023  }
```

lib/unistd/close.c

«

Si veda la sezione u0.7.

```
3810001  #include <unistd.h>
3810002  #include <errno.h>
3810003  #include <sys/os16.h>
3810004  #include <string.h>
3810005  //------------------------------------------------------------------------
3810006  int
3810007  close (int fdn)
3810008  {
3810009      sysmsg_close_t msg;
3810010      msg.fdn = fdn;
3810011      sys (SYS_CLOSE, &msg, (sizeof msg));
```

```
3810012      errno = msg.errno;
3810013      errrln = msg.errrln;
3810014      strncpy (errfn, msg.errfn, PATH_MAX);
3810015      return (msg.ret);
3810016  }
```

lib/unistd/dup.c

«

Si veda la sezione u0.8.

```
3820001  #include <unistd.h>
3820002  #include <sys/os16.h>
3820003  #include <string.h>
3820004  #include <errno.h>
3820005  //-------------------------------------------------------------------
3820006  int
3820007  dup (int fdn_old)
3820008  {
3820009      sysmsg_dup_t msg;
3820010      //
3820011      msg.fdn_old = fdn_old;
3820012      //
3820013      sys (SYS_DUP, &msg, (sizeof msg));
3820014      //
3820015      errno = msg.errno;
3820016      errrln = msg.errrln;
3820017      strncpy (errfn, msg.errfn, PATH_MAX);
3820018      return (msg.ret);
3820019  }
```

lib/unistd/dup2.c

«

Si veda la sezione u0.8.

```
3830001  #include <unistd.h>
3830002  #include <sys/os16.h>
3830003  #include <string.h>
3830004  #include <errno.h>
3830005  //-------------------------------------------------------------------
3830006  int
3830007  dup2 (int fdn_old, int fdn_new)
```

```
3830008 {
3830009     sysmsg_dup2_t msg;
3830010     //
3830011     msg.fdn_old = fdn_old;
3830012     msg.fdn_new = fdn_new;
3830013     //
3830014     sys (SYS_DUP2, &msg, (sizeof msg));
3830015     //
3830016     errno = msg.errno;
3830017     errln = msg.errln;
3830018     strncpy (errfn, msg.errfn, PATH_MAX);
3830019     return (msg.ret);
3830020 }
```

lib/unistd/environ.c

«

Si veda la sezione u0.1.

```
3840001 #include <unistd.h>
3840002 //------------------------------------------------------------------------
3840003 char **environ;
```

lib/unistd/execl.c

«

Si veda la sezione u0.20.

```
3850001 #include <unistd.h>
3850002 //------------------------------------------------------------------------
3850003 int
3850004 execl (const char *path, const char *arg, ...)
3850005 {
3850006     int   argc;
3850007     char *arg_next;
3850008     char *argv[ARG_MAX/2];
3850009     //
3850010     va_list ap;
3850011     va_start (ap, arg);
3850012     //
3850013     arg_next = arg;
3850014     //
3850015     for (argc = 0; argc < ARG_MAX/2; argc++)
```

```
3850016         {
3850017           argv[argc] = arg_next;
3850018           if (argv[argc] == NULL)
3850019             {
3850020               break;       // End of arguments.
3850021             }
3850022           arg_next = va_arg (ap, char *);
3850023         }
3850024       //
3850025       return (execve (path, argv, environ));       // [1]
3850026   }
3850027   //
3850028   // The variable 'environ' is declared as 'char **environ' and is
3850029   // included from <unistd.h>.
3850030   //
```

lib/unistd/execle.c

«

Si veda la sezione u0.20.

```
3860001   #include <unistd.h>
3860002   //---------------------------------------------------------------------
3860003   int
3860004   execle (const char *path, const char *arg, ...)
3860005   {
3860006       int    argc;
3860007       char  *arg_next;
3860008       char  *argv[ARG_MAX/2];
3860009       char **envp;
3860010       //
3860011       va_list ap;
3860012       va_start (ap, arg);
3860013       //
3860014       arg_next = arg;
3860015       //
3860016       for (argc = 0; argc < ARG_MAX/2; argc++)
3860017         {
3860018           argv[argc] = arg_next;
3860019           if (argv[argc] == NULL)
3860020             {
3860021               break;       // End of arguments.
3860022             }
```

```
3860023             arg_next = va_arg (ap, char *);
3860024           }
3860025         //
3860026         envp = va_arg (ap, char **);
3860027         //
3860028         return (execve (path, argv, envp));
3860029     }
```

lib/unistd/execlp.c

«

Si veda la sezione u0.20.

```
3870001 #include <unistd.h>
3870002 #include <string.h>
3870003 #include <stdlib.h>
3870004 #include <errno.h>
3870005 #include <sys/os16.h>
3870006 //-----------------------------------------------------------------------
3870007 int
3870008 execlp (const char *path, const char *arg, ...)
3870009 {
3870010     int   argc;
3870011     char *arg_next;
3870012     char *argv[ARG_MAX/2];
3870013     char  command[PATH_MAX];
3870014     int   status;
3870015     //
3870016     va_list ap;
3870017     va_start (ap, arg);
3870018     //
3870019     arg_next = arg;
3870020     //
3870021     for (argc = 0; argc < ARG_MAX/2; argc++)
3870022       {
3870023         argv[argc] = arg_next;
3870024         if (argv[argc] == NULL)
3870025           {
3870026             break;      // End of arguments.
3870027           }
3870028         arg_next = va_arg (ap, char *);
3870029       }
3870030     //
```

```
3870031      // Get a full command path if necessary.
3870032      //
3870033      status = namep (path, command, (size_t) PATH_MAX);
3870034      if (status != 0)
3870035        {
3870036          //
3870037          // Variable 'errno' is already set by 'commandp()'.
3870038          //
3870039          return (-1);
3870040        }
3870041      //
3870042      // Return calling 'execve()'
3870043      //
3870044      return (execve (command, argv, environ));   // [1]
3870045  }
3870046  //
3870047  // The variable 'environ' is declared as 'char **environ' and is
3870048  // included from <unistd.h>.
3870049  //
```

lib/unistd/execv.c

«

Si veda la sezione u0.20.

```
3880001  #include <unistd.h>
3880002  //---------------------------------------------------------------------
3880003  int
3880004  execv (const char *path, char *const argv[])
3880005  {
3880006      return (execve (path, argv, environ));      // [1]
3880007  }
3880008  //
3880009  // The variable 'environ' is declared as 'char **environ' and is
3880010  // included from <unistd.h>.
3880011  //
```

«

Si veda la sezione u0.10.

```
3890001  #include <unistd.h>
3890002  #include <sys/types.h>
3890003  #include <sys/os16.h>
3890004  #include <errno.h>
3890005  #include <string.h>
3890006  #include <string.h>
3890007  //-----------------------------------------------------------------------
3890008  int
3890009  execve (const char *path, char *const argv[], char *const envp[])
3890010  {
3890011      sysmsg_exec_t msg;
3890012      size_t         size;
3890013      size_t         arg_size;
3890014      int            argc;
3890015      size_t         env_size;
3890016      int            envc;
3890017      char          *arg_data = msg.arg_data;
3890018      char          *env_data = msg.env_data;
3890019      //
3890020      msg.ret   = 0;
3890021      msg.errno = 0;
3890022      //
3890023      strncpy (msg.path, path, PATH_MAX);
3890024      //
3890025      // Copy 'argv[]' inside a the message buffer 'msg.arg_data',
3890026      // separating each string with a null character and counting the
3890027      // number of strings inside 'argc'.
3890028      //
3890029      for (argc = 0, arg_size = 0, size = 0;
3890030           argv       != NULL         &&
3890031           argc        < (ARG_MAX/16) &&
3890032           arg_size    < ARG_MAX/2    &&
3890033           argv[argc] != NULL;
3890034           argc++, arg_size += size)
3890035        {
3890036          size = strlen (argv[argc]);
3890037          size++;          // Count also the final null character.
3890038          if (size > (ARG_MAX/2 - arg_size))
3890039            {
3890040              errset (E2BIG);     // Argument list too long.
```

```
3890041             return (-1);
3890042           }
3890043         strncpy (arg_data, argv[argc], size);
3890044         arg_data += size;
3890045       }
3890046     msg.argc = argc;
3890047     //
3890048     // Copy 'envp[]' inside a the message buffer 'msg.env_data',
3890049     // separating each string with a null character and counting the
3890050     // number of strings inside 'envc'.
3890051     //
3890052     for (envc = 0, env_size = 0, size = 0;
3890053          envp       != NULL         &&
3890054          envc        < (ARG_MAX/16) &&
3890055          env_size    < ARG_MAX/2    &&
3890056          envp[envc] != NULL;
3890057          envc++, env_size += size)
3890058       {
3890059         size = strlen (envp[envc]);
3890060         size++;         // Count also the final null character.
3890061         if (size > (ARG_MAX/2 - env_size))
3890062           {
3890063             errset (E2BIG);     // Argument list too long.
3890064             return (-1);
3890065           }
3890066         strncpy (env_data, envp[envc], size);
3890067         env_data += size;
3890068       }
3890069     msg.envc = envc;
3890070     //
3890071     // System call.
3890072     //
3890073     sys (SYS_EXEC, &msg, (sizeof msg));
3890074     //
3890075     // Should not return, but if it does, then there is an error.
3890076     //
3890077     errno = msg.errno;
3890078     errln = msg.errln;
3890079     strncpy (errfn, msg.errfn, PATH_MAX);
3890080     return (msg.ret);
3890081 }
```

lib/unistd/execvp.c

«

Si veda la sezione u0.20.

```
3900001 #include <unistd.h>
3900002 #include <string.h>
3900003 #include <stdlib.h>
3900004 #include <errno.h>
3900005 #include <sys/os16.h>
3900006 //---------------------------------------------------------------------
3900007 int
3900008 execvp (const char *path, char *const argv[])
3900009 {
3900010     char  command[PATH_MAX];
3900011     int   status;
3900012     //
3900013     // Get a full command path if necessary.
3900014     //
3900015     status = namep (path, command, (size_t) PATH_MAX);
3900016     if (status != 0)
3900017       {
3900018         //
3900019         // Variable 'errno' is already set by 'namep()'.
3900020         //
3900021         return (-1);
3900022       }
3900023     //
3900024     // Return calling 'execve()'
3900025     //
3900026     return (execve (command, argv, environ));   // [1]
3900027 }
3900028 //
3900029 // The variable 'environ' is declared as 'char **environ' and is
3900030 // included from <unistd.h>.
3900031 //
```

lib/unistd/fchdir.c

«

Si veda la sezione u0.2.

```
3910001 #include <unistd.h>
3910002 #include <errno.h>
3910003 //---------------------------------------------------------------------
```

```
3910004 int
3910005 fchdir (int fdn)
3910006 {
3910007     //
3910008     // os16 requires to keep track of the path for the current working
3910009     // directory. The standard function 'fchdir()' is not applicable.
3910010     //
3910011     errset (E_NOT_IMPLEMENTED);
3910012     return (-1);
3910013 }
```

lib/unistd/fchown.c

«

Si veda la sezione u0.5.

```
3920001 #include <unistd.h>
3920002 #include <string.h>
3920003 #include <const.h>
3920004 #include <sys/os16.h>
3920005 #include <errno.h>
3920006 #include <limits.h>
3920007 //------------------------------------------------------------------------
3920008 int
3920009 fchown (int fdn, uid_t uid, gid_t gid)
3920010 {
3920011     sysmsg_fchown_t msg;
3920012     //
3920013     msg.fdn = fdn;
3920014     msg.uid = uid;
3920015     msg.gid = gid;
3920016     //
3920017     sys (SYS_FCHOWN, &msg, (sizeof msg));
3920018     //
3920019     errno = msg.errno;
3920020     errln = msg.errln;
3920021     strncpy (errfn, msg.errfn, PATH_MAX);
3920022     return (msg.ret);
3920023 }
```

«

Si veda la sezione u0.14.

```
3930001  #include <unistd.h>
3930002  #include <sys/types.h>
3930003  #include <sys/os16.h>
3930004  #include <errno.h>
3930005  #include <string.h>
3930006  //------------------------------------------------------------------------
3930007  pid_t
3930008  fork (void)
3930009  {
3930010      sysmsg_fork_t msg;
3930011      //
3930012      // Set the return value for the child process.
3930013      //
3930014      msg.ret = 0;
3930015      //
3930016      // Do the system call.
3930017      //
3930018      sys (SYS_FORK, &msg, (sizeof msg));
3930019      //
3930020      // If the system call has successfully generated a copy of
3930021      // the original process, the following code is executed from
3930022      // the parent and the child. But the child has the 'msg'
3930023      // structure untouched, while the parent has, at least, the
3930024      // pid number inside 'msg.ret'.
3930025      // If the system call fails, there is no child, and the
3930026      // parent finds the return value equal to -1, with an
3930027      // error number.
3930028      //
3930029      errno = msg.errno;
3930030      errln = msg.errln;
3930031      strncpy (errfn, msg.errfn, PATH_MAX);
3930032      return (msg.ret);
3930033  }
```

«

Si veda la sezione u0.16.

```
3940001 #include <unistd.h>
3940002 #include <sys/types.h>
3940003 #include <sys/os16.h>
3940004 #include <errno.h>
3940005 #include <stddef.h>
3940006 #include <string.h>
3940007 //------------------------------------------------------------------------
3940008 char *
3940009 getcwd (char *buffer, size_t size)
3940010 {
3940011     sysmsg_uarea_t msg;
3940012     //
3940013     // Check arguments: the buffer must be given.
3940014     //
3940015     if (buffer == NULL)
3940016       {
3940017         errset (EINVAL);
3940018         return (NULL);
3940019       }
3940020     //
3940021     // Make shure that the last character, inside the working directory
3940022     // path is a null character.
3940023     //
3940024     msg.path_cwd[PATH_MAX-1] = 0;
3940025     //
3940026     // Just get the user area data.
3940027     //
3940028     sys (SYS_UAREA, &msg, (sizeof msg));
3940029     //
3940030     // Check that the path is still correctly terminated. If it isn't,
3940031     // the path is longer than the implementation limits, and it is
3940032     // really *bad*.
3940033     //
3940034     if (msg.path_cwd[PATH_MAX-1] != 0)
3940035       {
3940036         errset (E_LIMIT);       // Exceeded implementation limits.
3940037         return (NULL);
3940038       }
3940039     //
3940040     // If the path is larger than the buffer size, return an error.
```

```
3940041      // Please note that the parameter 'size' must include the
3940042      // terminating null character, so, if the string is equal to
3940043      // the size, it is already beyond the size limit.
3940044      //
3940045      if (strlen (msg.path_cwd) >= size)
3940046        {
3940047          errset (ERANGE);        // Result too large.
3940048          return (NULL);
3940049        }
3940050      //
3940051      // Everything is fine, so, copy the path to the buffer and return.
3940052      //
3940053      strncpy (buffer, msg.path_cwd, size);
3940054      return (buffer);
3940055  }
```

lib/unistd/geteuid.c

«

Si veda la sezione u0.18.

```
3950001  #include <unistd.h>
3950002  #include <sys/types.h>
3950003  #include <sys/os16.h>
3950004  #include <errno.h>
3950005  //------------------------------------------------------------------------
3950006  uid_t
3950007  geteuid (void)
3950008  {
3950009      sysmsg_uarea_t msg;
3950010      sys (SYS_UAREA, &msg, (sizeof msg));
3950011      return (msg.euid);
3950012  }
```

lib/unistd/getopt.c

«

Si veda la sezione u0.52.

```
3960001  #include <unistd.h>
3960002  #include <sys/types.h>
3960003  #include <sys/os16.h>
3960004  #include <errno.h>
```

```
3960005 //-----------------------------------------------------------------------
3960006 char *optarg;
3960007 int   optind    = 1;
3960008 int   opterr    = 1;
3960009 int   optopt    = 0;
3960010 //-----------------------------------------------------------------------
3960011 static void getopt_no_argument (int opt);
3960012 //-----------------------------------------------------------------------
3960013 int
3960014 getopt (int argc, char *const argv[], const char *optstring)
3960015 {
3960016     static int o = 0;           // Index to scan grouped options.
3960017            int s;               // Index to scan 'optstring'
3960018            int opt;             // Current option letter.
3960019            int flag_argument;   // If there should be an argument.
3960020     //
3960021     // Entering the function, 'flag_argument' is zero. Just to make
3960022     // it clear:
3960023     //
3960024     flag_argument = 0;
3960025     //
3960026     // Scan 'argv[]' elements, starting form the value that 'optind'
3960027     // already have.
3960028     //
3960029     for (; optind < argc; optind++)
3960030       {
3960031         //
3960032         // If an option is expected, some check must be done at
3960033         // the beginning.
3960034         //
3960035         if (!flag_argument)
3960036           {
3960037             //
3960038             // Check if the scan is finished and 'optind' should be kept
3960039             // untouched:
3960040             //   'argv[optind]'    is a null pointer;
3960041             //   'argv[optind][0]' is not the character '-';
3960042             //   'argv[optind]'    points to the string "-";
3960043             //   all 'argv[]' elements are parsed.
3960044             //
3960045             if (argv[optind] == NULL
3960046              || argv[optind][0] != '-'
3960047              || argv[optind][1] == 0
```

```
3960048                || optind >= argc)
3960049                 {
3960050                   return (-1);
3960051                 }
3960052               //
3960053               // Check if the scan is finished and 'optind' is to be
3960054               // incremented:
3960055               //   'argv[optind]'    points to the string "--".
3960056               //
3960057               if (argv[optind][0] == '-'
3960058                && argv[optind][1] == '-'
3960059                && argv[optind][2] == 0)
3960060                 {
3960061                   optind++;
3960062                   return (-1);
3960063                 }
3960064             }
3960065           //
3960066           // Scan 'argv[optind]' using the static index 'o'.
3960067           //
3960068           for (; o < strlen (argv[optind]); o++)
3960069             {
3960070               //
3960071               // If there should be an option, index 'o' should
3960072               // start from 1, because 'argv[optind][0]' must
3960073               // be equal to '-'.
3960074               //
3960075               if (!flag_argument && (o == 0))
3960076                 {
3960077                   //
3960078                   // As there is no options, 'o' cannot start
3960079                   // from zero, so a new loop is done.
3960080                   //
3960081                   continue;
3960082                 }
3960083               //
3960084               if (flag_argument)
3960085                 {
3960086                   //
3960087                   // There should be an argument, starting from
3960088                   // 'argv[optind][o]'.
3960089                   //
3960090                   if ((o == 0) && (argv[optind][o] == '-'))
```

```
3960091                     {
3960092                       //
3960093                       // `argv[optind][0]' is equal to '-', but there
3960094                       // should be an argument instead: the argument
3960095                       // is missing.
3960096                       //
3960097                       optarg = NULL;
3960098                       //
3960099                       if (optstring[0] == ':')
3960100                         {
3960101                           //
3960102                           // As the option string starts with `:' the
3960103                           // function must return `:'.
3960104                           //
3960105                           optopt = opt;
3960106                           opt = ':';
3960107                         }
3960108                       else
3960109                         {
3960110                           //
3960111                           // As the option string does not start with `:'
3960112                           // the function must return `?'.
3960113                           //
3960114                           getopt_no_argument (opt);
3960115                           optopt = opt;
3960116                           opt = '?';
3960117                         }
3960118                       //
3960119                       // `optind' is left untouched.
3960120                       //
3960121                     }
3960122                   else
3960123                     {
3960124                       //
3960125                       // The argument is found: `optind' is to be
3960126                       // incremented and `o' is reset.
3960127                       //
3960128                       optarg = &argv[optind][o];
3960129                       optind++;
3960130                       o = 0;
3960131                     }
3960132                   //
3960133                   // Return the option, or ':', or '?'.
```

```
3960134                     //
3960135                     return (opt);
3960136                   }
3960137                 else
3960138                   {
3960139                     //
3960140                     // It should be an option: 'optstring[]' must be
3960141                     // scanned.
3960142                     //
3960143                     opt = argv[optind][o];
3960144                     //
3960145                     for (s = 0, optopt = 0; s < strlen (optstring); s++)
3960146                       {
3960147                         //
3960148                         // If 'optsting[0]' is equal to ':', index 's' must
3960149                         // start at 1.
3960150                         //
3960151                         if ((s == 0) && (optstring[0] == ':'))
3960152                           {
3960153                             continue;
3960154                           }
3960155                         //
3960156                         if (opt == optstring[s])
3960157                           {
3960158                             //
3960159                             if (optstring[s+1] == ':')
3960160                               {
3960161                                 //
3960162                                 // There is an argument.
3960163                                 //
3960164                                 flag_argument = 1;
3960165                                 break;
3960166                               }
3960167                             else
3960168                               {
3960169                                 //
3960170                                 // There is no argument.
3960171                                 //
3960172                                 o++;
3960173                                 return (opt);
3960174                               }
3960175                           }
3960176                       }
```

```
3960177                   //
3960178                   if (s >= strlen (optstring))
3960179                     {
3960180                       //
3960181                       // The 'optstring' scan is concluded with no
3960182                       // match.
3960183                       //
3960184                       o++;
3960185                       optopt = opt;
3960186                       return ('?');
3960187                     }
3960188                   //
3960189                   // Otherwise the loop was broken.
3960190                   //
3960191                 }
3960192             }
3960193           //
3960194           // Check index 'o'.
3960195           //
3960196           if (o >= strlen (argv[optind]))
3960197             {
3960198               //
3960199               // There are no more options or there is no argument
3960200               // inside current 'argv[optind]' string. Index 'o' is
3960201               // reset before the next loop.
3960202               //
3960203               o = 0;
3960204             }
3960205         }
3960206       //
3960207       // No more elements inside 'argv' or loop broken: there might be a
3960208       // missing argument.
3960209       //
3960210       if (flag_argument)
3960211         {
3960212           //
3960213           // Missing option argument.
3960214           //
3960215           optarg = NULL;
3960216           //
3960217           if (optstring[0] == ':')
3960218             {
3960219               return (':');
```

```
3960220             }
3960221           else
3960222             {
3960223               getopt_no_argument (opt);
3960224               return ('?');
3960225             }
3960226         }
3960227       //
3960228       return (-1);
3960229   }
3960230   //-----------------------------------------------------------------------
3960231   static void
3960232   getopt_no_argument (int opt)
3960233   {
3960234       if (opterr)
3960235         {
3960236           fprintf (stderr, "Missing argument for option `-%c'\n", opt);
3960237         }
3960238   }
```

lib/unistd/getpgrp.c

«

Si veda la sezione u0.20.

```
3970001   #include <unistd.h>
3970002   #include <sys/types.h>
3970003   #include <sys/os16.h>
3970004   #include <errno.h>
3970005   //-----------------------------------------------------------------------
3970006   pid_t
3970007   getpgrp (void)
3970008   {
3970009       sysmsg_uarea_t msg;
3970010       sys (SYS_UAREA, &msg, (sizeof msg));
3970011       return (msg.pgrp);
3970012   }
```

lib/unistd/getpid.c

«

Si veda la sezione u0.20.

```
3980001  #include <unistd.h>
3980002  #include <sys/types.h>
3980003  #include <sys/os16.h>
3980004  #include <errno.h>
3980005  //---------------------------------------------------------------------
3980006  pid_t
3980007  getpid (void)
3980008  {
3980009      sysmsg_uarea_t msg;
3980010      sys (SYS_UAREA, &msg, (sizeof msg));
3980011      return (msg.pid);
3980012  }
```

lib/unistd/getppid.c

«

Si veda la sezione u0.20.

```
3990001  #include <unistd.h>
3990002  #include <sys/types.h>
3990003  #include <sys/os16.h>
3990004  #include <errno.h>
3990005  //---------------------------------------------------------------------
3990006  pid_t
3990007  getppid (void)
3990008  {
3990009      sysmsg_uarea_t msg;
3990010      sys (SYS_UAREA, &msg, (sizeof msg));
3990011      return (msg.ppid);
3990012  }
```

lib/unistd/getuid.c

«

Si veda la sezione u0.18.

```
4000001  #include <unistd.h>
4000002  #include <sys/types.h>
4000003  #include <sys/os16.h>
4000004  #include <errno.h>
```

```
4000005 //------------------------------------------------------------------------
4000006 uid_t
4000007 getuid (void)
4000008 {
4000009     sysmsg_uarea_t msg;
4000010     sys (SYS_UAREA, &msg, (sizeof msg));
4000011     return (msg.uid);
4000012 }
```

lib/unistd/isatty.c

«

Si veda la sezione u0.61.

```
4010001 #include <sys/stat.h>
4010002 #include <sys/os16.h>
4010003 #include <unistd.h>
4010004 #include <sys/types.h>
4010005 #include <errno.h>
4010006 //------------------------------------------------------------------------
4010007 int
4010008 isatty (int fdn)
4010009 {
4010010     struct stat file_status;
4010011     //
4010012     // Verify to have valid input data.
4010013     //
4010014     if (fdn < 0)
4010015       {
4010016         errset (EBADF);
4010017         return (0);
4010018       }
4010019     //
4010020     // Verify the standard input.
4010021     //
4010022     if (fstat(fdn, &file_status) == 0)
4010023       {
4010024         if (major (file_status.st_rdev) == DEV_CONSOLE_MAJOR)
4010025           {
4010026             return (1); // Meaning it is ok!
4010027           }
4010028         if (major (file_status.st_rdev) == DEV_TTY_MAJOR)
4010029           {
```

```
4010030             return (1); // Meaning it is ok!
4010031           }
4010032       }
4010033     else
4010034       {
4010035         errset (errno);
4010036         return (0);
4010037       }
4010038     //
4010039     // If here, it is not a terminal of any kind.
4010040     //
4010041     errset (EINVAL);
4010042     return (0);
4010043 }
```

lib/unistd/link.c

«

Si veda la sezione u0.23.

```
4020001 #include <unistd.h>
4020002 #include <string.h>
4020003 #include <const.h>
4020004 #include <sys/os16.h>
4020005 #include <errno.h>
4020006 #include <limits.h>
4020007 //-----------------------------------------------------------------------
4020008 int
4020009 link (const char *path_old, const char *path_new)
4020010 {
4020011     sysmsg_link_t msg;
4020012     //
4020013     strncpy (msg.path_old, path_old, PATH_MAX);
4020014     strncpy (msg.path_new, path_new, PATH_MAX);
4020015     //
4020016     sys (SYS_LINK, &msg, (sizeof msg));
4020017     //
4020018     errno = msg.errno;
4020019     errln = msg.errln;
4020020     strncpy (errfn, msg.errfn, PATH_MAX);
4020021     return (msg.ret);
4020022 }
```

lib/unistd/lseek.c

«

Si veda la sezione u0.24.

```
4030001  #include <unistd.h>
4030002  #include <sys/types.h>
4030003  #include <sys/os16.h>
4030004  #include <errno.h>
4030005  #include <string.h>
4030006  //------------------------------------------------------------------------
4030007  off_t
4030008  lseek (int fdn, off_t offset, int whence)
4030009  {
4030010      sysmsg_lseek_t  msg;
4030011      msg.fdn    = fdn;
4030012      msg.offset = offset;
4030013      msg.whence = whence;
4030014      sys (SYS_LSEEK, &msg, (sizeof msg));
4030015      errno = msg.errno;
4030016      errln = msg.errln;
4030017      strncpy (errfn, msg.errfn, PATH_MAX);
4030018      return (msg.ret);
4030019  }
```

lib/unistd/read.c

«

Si veda la sezione u0.29.

```
4040001  #include <unistd.h>
4040002  #include <sys/os16.h>
4040003  #include <errno.h>
4040004  #include <string.h>
4040005  #include <stdio.h>
4040006  //------------------------------------------------------------------------
4040007  ssize_t
4040008  read (int fdn, void *buffer, size_t count)
4040009  {
4040010      sysmsg_read_t msg;
4040011      //
4040012      // Reduce size of read if necessary.
4040013      //
4040014      if (count > BUFSIZ)
4040015        {
```

```
4040016           count = BUFSIZ;
4040017         }
4040018       //
4040019       // Fill the message.
4040020       //
4040021       msg.fdn   = fdn;
4040022       msg.count = count;
4040023       msg.eof   = 0;
4040024       msg.ret   = 0;
4040025       //
4040026       // Repeat syscall, until something is received or end of file is
4040027       // reached.
4040028       //
4040029       while (1)
4040030         {
4040031           sys (SYS_READ, &msg, (sizeof msg));
4040032           if (msg.ret != 0 || msg.eof)
4040033             {
4040034               break;
4040035             }
4040036         }
4040037       //
4040038       // Before return: be careful with the 'msg.buffer' copy, because
4040039       // it cannot be longer than 'count', otherwise, some unexpected
4040040       // memory will be overwritten!
4040041       //
4040042       if (msg.ret < 0)
4040043         {
4040044           //
4040045           // No valid read, no change inside the buffer.
4040046           //
4040047           errno = msg.errno;
4040048           return (msg.ret);
4040049         }
4040050       //
4040051       if (msg.ret > count)
4040052         {
4040053           //
4040054           // A strange value was returned. Considering it a read error.
4040055           //
4040056           errset (EIO);                    // I/O error.
4040057           return (-1);
4040058         }
```

```
4040059       //
4040060       // A valid read: fill the buffer with 'msg.ret' bytes.
4040061       //
4040062       memcpy (buffer, msg.buffer, msg.ret);
4040063       //
4040064       // Return.
4040065       //
4040066       return (msg.ret);
4040067   }
```

lib/unistd/rmdir.c

«

Si veda la sezione u0.30.

```
4050001   #include <unistd.h>
4050002   #include <string.h>
4050003   #include <const.h>
4050004   #include <sys/os16.h>
4050005   #include <errno.h>
4050006   #include <limits.h>
4050007   //-------------------------------------------------------------------
4050008   int
4050009   rmdir (const char *path)
4050010   {
4050011       sysmsg_stat_t   msg_stat;
4050012       sysmsg_unlink_t msg_unlink;
4050013       //
4050014       if (path == NULL)
4050015         {
4050016           errset (EINVAL);
4050017           return (-1);
4050018         }
4050019       //
4050020       strncpy (msg_stat.path, path, PATH_MAX);
4050021       //
4050022       sys (SYS_STAT, &msg_stat, (sizeof msg_stat));
4050023       //
4050024       if (msg_stat.ret != 0)
4050025         {
4050026           errno = msg_stat.errno;
4050027           errln = msg_stat.errln;
4050028           strncpy (errfn, msg_stat.errfn, PATH_MAX);
```

```
4050029           return (msg_stat.ret);
4050030         }
4050031       //
4050032       if (!S_ISDIR (msg_stat.stat.st_mode))
4050033         {
4050034           errset (ENOTDIR);               // Not a directory.
4050035           return (-1);
4050036         }
4050037       //
4050038       strncpy (msg_unlink.path, path, PATH_MAX);
4050039       //
4050040       sys (SYS_UNLINK, &msg_unlink, (sizeof msg_unlink));
4050041       //
4050042       errno = msg_unlink.errno;
4050043       errln = msg_unlink.errln;
4050044       strncpy (errfn, msg_unlink.errfn, PATH_MAX);
4050045       return (msg_unlink.ret);
4050046   }
```

lib/unistd/seteuid.c

«

Si veda la sezione u0.33.

```
4060001   #include <unistd.h>
4060002   #include <sys/types.h>
4060003   #include <sys/os16.h>
4060004   #include <errno.h>
4060005   #include <string.h>
4060006   //-----------------------------------------------------------------------
4060007   int
4060008   seteuid (uid_t uid)
4060009   {
4060010       sysmsg_seteuid_t msg;
4060011       msg.ret   = 0;
4060012       msg.errno = 0;
4060013       msg.euid  = uid;
4060014       sys (SYS_SETEUID, &msg, (sizeof msg));
4060015       errno = msg.errno;
4060016       errln = msg.errln;
4060017       strncpy (errfn, msg.errfn, PATH_MAX);
4060018       return (msg.ret);
4060019   }
```

```
4060020
```

lib/unistd/setpgrp.c

« Si veda la sezione u0.32.

```
4070001  #include <unistd.h>
4070002  #include <sys/os16.h>
4070003  #include <stddef.h>
4070004  //------------------------------------------------------------------------
4070005  int
4070006  setpgrp (void)
4070007  {
4070008      sys (SYS_PGRP, NULL, (size_t) 0);
4070009      return (0);
4070010  }
```

lib/unistd/setuid.c

« Si veda la sezione u0.33.

```
4080001  #include <unistd.h>
4080002  #include <sys/types.h>
4080003  #include <sys/os16.h>
4080004  #include <errno.h>
4080005  #include <string.h>
4080006  //------------------------------------------------------------------------
4080007  int
4080008  setuid (uid_t uid)
4080009  {
4080010      sysmsg_setuid_t msg;
4080011      msg.ret   = 0;
4080012      msg.errno = 0;
4080013      msg.euid  = uid;
4080014      sys (SYS_SETUID, &msg, (sizeof msg));
4080015      errno = msg.errno;
4080016      errln = msg.errln;
4080017      strncpy (errfn, msg.errfn, PATH_MAX);
4080018      return (msg.ret);
4080019  }
```

«

Si veda la sezione u0.35.

```
4090001  #include <unistd.h>
4090002  #include <sys/types.h>
4090003  #include <sys/os16.h>
4090004  #include <errno.h>
4090005  #include <time.h>
4090006  //---------------------------------------------------------------------
4090007  unsigned int
4090008  sleep (unsigned int seconds)
4090009  {
4090010      sysmsg_sleep_t  msg;
4090011      time_t          start;
4090012      time_t          end;
4090013      int             slept;
4090014      //
4090015      if (seconds == 0)
4090016        {
4090017          return (0);
4090018        }
4090019      //
4090020      msg.events  = WAKEUP_EVENT_TIMER;
4090021      msg.seconds = seconds;
4090022      sys (SYS_SLEEP, &msg, (sizeof msg));
4090023      start = msg.ret;
4090024      end   = time (NULL);
4090025      slept = end - msg.ret;
4090026      //
4090027      if (slept < 0)
4090028        {
4090029          return (seconds);
4090030        }
4090031      else if (slept < seconds)
4090032        {
4090033          return (seconds - slept);
4090034        }
4090035      else
4090036        {
4090037          return (0);
4090038        }
4090039  }
```

«

Si veda la sezione u0.124.

```
4100001 #include <sys/os16.h>
4100002 #include <sys/stat.h>
4100003 #include <unistd.h>
4100004 #include <sys/types.h>
4100005 #include <errno.h>
4100006 #include <limits.h>
4100007 //---------------------------------------------------------------------
4100008 char *
4100009 ttyname (int fdn)
4100010 {
4100011     int          dev_minor;
4100012     struct stat file_status;
4100013     static char name[PATH_MAX];
4100014     //
4100015     // Verify to have valid input data.
4100016     //
4100017     if (fdn < 0)
4100018       {
4100019         errset (EBADF);
4100020         return (NULL);
4100021       }
4100022     //
4100023     // Verify the file descriptor.
4100024     //
4100025     if (fstat (fdn, &file_status) == 0)
4100026       {
4100027         if (major (file_status.st_rdev) == DEV_CONSOLE_MAJOR)
4100028           {
4100029             dev_minor = minor (file_status.st_rdev);
4100030             //
4100031             // If minor is equal to 0xFF, it is '/dev/console'.
4100032             //
4100033             if (dev_minor < 0xFF)
4100034               {
4100035                 sprintf (name, "/dev/console%i", dev_minor);
4100036               }
4100037             else
4100038               {
4100039                 strcpy (name, "/dev/console");
4100040               }
```

```
4100041             return (name);
4100042           }
4100043         else if (file_status.st_rdev == DEV_TTY)
4100044           {
4100045             strcpy (name, "/dev/tty");
4100046             return (name);
4100047           }
4100048         else
4100049           {
4100050             errset (ENOTTY);
4100051             return (NULL);
4100052           }
4100053       }
4100054     else
4100055       {
4100056         errset (errno);
4100057         return (NULL);
4100058       }
4100059 }
```

lib/unistd/unlink.c

«

Si veda la sezione u0.42.

```
4110001 #include <unistd.h>
4110002 #include <string.h>
4110003 #include <const.h>
4110004 #include <sys/os16.h>
4110005 #include <errno.h>
4110006 #include <limits.h>
4110007 //------------------------------------------------------------------------
4110008 int
4110009 unlink (const char *path)
4110010 {
4110011     sysmsg_unlink_t msg;
4110012     //
4110013     strncpy (msg.path, path, PATH_MAX);
4110014     //
4110015     sys (SYS_UNLINK, &msg, (sizeof msg));
4110016     //
4110017     errno = msg.errno;
4110018     errln = msg.errln;
```

```
4110019     strncpy (errfn, msg.errfn, PATH_MAX);
4110020     return (msg.ret);
4110021 }
```

lib/unistd/write.c

«

Si veda la sezione u0.44.

```
4120001 #include <unistd.h>
4120002 #include <sys/os16.h>
4120003 #include <errno.h>
4120004 #include <string.h>
4120005 #include <const.h>
4120006 #include <stdio.h>
4120007 //------------------------------------------------------------------------
4120008 ssize_t
4120009 write (int fdn, const void *buffer, size_t count)
4120010 {
4120011     sysmsg_write_t msg;
4120012     //
4120013     // Reduce size of write if necessary.
4120014     //
4120015     if (count > BUFSIZ)
4120016       {
4120017         count = BUFSIZ;
4120018       }
4120019     //
4120020     // Fill the message.
4120021     //
4120022     msg.fdn   = fdn;
4120023     msg.count = count;
4120024     memcpy (msg.buffer, buffer, count);
4120025     //
4120026     // Syscall.
4120027     //
4120028     sys (SYS_WRITE, &msg, (sizeof msg));
4120029     //
4120030     // Check result and return.
4120031     //
4120032     if (msg.ret < 0)
4120033       {
4120034         //
```

```
4120035           // No valid read, no change inside the buffer.
4120036           //
4120037           errno = msg.errno;
4120038           errln = msg.errln;
4120039           strncpy (errfn, msg.errfn, PATH_MAX);
4120040           return (msg.ret);
4120041         }
4120042       //
4120043       if (msg.ret > count)
4120044         {
4120045           //
4120046           // A strange value was returned. Considering it a read error.
4120047           //
4120048           errset (EIO);                    // I/O error.
4120049           return (-1);
4120050         }
4120051       //
4120052       // A valid write return.
4120053       //
4120054       return (msg.ret);
4120055   }
```

os16: «lib/utime.h»

«

Si veda la sezione u0.2.

```
4130001   #ifndef _UTIME_H
4130002   #define _UTIME_H        1
4130003
4130004   #include <const.h>
4130005   #include <restrict.h>
4130006   #include <sys/types.h>          // time_t
4130007
4130008   //---------------------------------------------------------------------
4130009   struct utimbuf {
4130010       time_t  actime;
4130011       time_t  modtime;
4130012   };
4130013   //---------------------------------------------------------------------
4130014   int utime (const char *path, const struct utimbuf *times);
4130015   //---------------------------------------------------------------------
4130016
```

```
4130017  #endif
```

lib/utime/utime.c

«

Si veda la sezione u0.2.

```
4140001  #include <utime.h>
4140002  #include <errno.h>
4140003  //------------------------------------------------------------------------
4140004  int
4140005  utime (const char *path, const struct utimbuf *times)
4140006  {
4140007      //
4140008      // Currently not implemented.
4140009      //
4140010      return (0);
4140011  }
```