In generale, la libreria offerta da un compilatore del linguaggio C si può estendere in modo imprecisato verso le definizioni dello standard POSIX. Per esempio, è normale che una libreria C includa le funzionalità relative alla gestione delle espressioni regolari, definite dallo standard POSIX. Pertanto, non esiste propriamente una libreria

C e una POSIX, va quindi verificato con il proprio compilatore cosa offrono effettivamente le librerie disponibili, specificando eventualmente, in fase di compilazione, l'inclusione di questa o quella libreria precompilata per la gestione di quella certa funzionalità POSIX.

Nei capitoli successivi vengono descritti alcuni dei file di intestazione previsti dallo standard POSIX, che a loro volta non sono già presi in considerazione dallo standard del linguaggio C. In certi casi viene mostrato come potrebbero essere realizzati questi file (gli esempi dovrebbero essere disponibili a partire da *allegati/c/*).

Tabella 70.1. Alcuni file di intestazione dello standard POSIX che non si trovano già nello standard del linguaggio C.

Intestazione	Descrizione	Riferimenti
`sys/types.h`	Tipi di dati derivati.	sezione 70.1
`sys/stat.h`	Definizione dei dati restituiti dalla funzione ***stat()***, necessari alla qualificazione delle caratteristiche dei file, e di alcune funzioni relative alla questione.	sezione 70.2
`strings.h`	Funzioni per il trattamento delle stringhe e simili che non sono già incluse in '`string.h`'.	sezione 70.3
`fcntl.h`	Opzioni per il controllo dei file, gestiti in qualità di descrittori.	sezione 70.4
`unistd.h`	Macro-variabili standard e molte funzioni sulla gestione dei file.	sezione 70.5
`dirent.h`	Gestione delle directory, in qualità di flussi, attraverso puntatori di tipo '**`DIR *`**'.	sezione 70.6

Intestazione	Descrizione	Riferimenti
`termios.h`	Configurazione dei dispositivi di terminale.	sezione 70.7

70.1 File «sys/types.h»

«

Il file ‘`sys/types.h`’ viene usato dallo standard POSIX per definire tutti i tipi di dati derivati, inclusi alcuni che già fanno parte dello standard C puro e semplice (si veda eventualmente la realizzazione di questo file nei sorgenti di os32, sezione 95.26). La tabella successiva ne descrive solo una parte.

Tipo derivato	Descrizione
`blkcnt_t`	Numero intero con segno. Quantità di blocchi, riferita ai file.
`blksize_t`	Numero intero con segno. Dimensione in blocchi di un file.
`clock_t`	Numero intero. Unità di tempo che rappresenta un ciclo virtuale di CPU. È già definito nel file ‘`time.h`’.
`dev_t`	Numero intero. Numero identificativo di un dispositivo.
`gid_t`	Numero intero senza segno. Numero identificativo del gruppo a cui appartiene un file.
`ino_t`	Numero intero senza segno. Numero di inode, ovvero il numero identificativo di un file.
`mode_t`	Numero intero. Attributo di accesso di un file.
`nlink_t`	Numero intero senza segno. Quantità di collegamenti fisici riferiti a un file.

Tipo derivato	Descrizione
`off_t`	Numero intero con segno. Scostamento relativo al contenuto di un file.
`pid_t`	Numero intero con segno. Numero che identifica un processo elaborativo. Il segno è necessario perché il valore −1 rappresenta un errore nella creazione di un processo.
`pthread_t`	Numero intero. Numero identificativo di un thread.
`size_t`	Numero intero senza segno. Usato per esprimere la grandezza di oggetti rappresentati in memoria (variabili, distanza tra puntatori e simili). Nello standard C, questo tipo derivato viene definito nel file '`stddef.h`'.
`ssize_t`	Numero intero con segno (*signed size_t*). Usato per esprimere una quantità di byte, se positivo, oppure un errore se negativo.
`time_t`	Numero intero con segno. Usato per esprimere un tempo in secondi, trascorso a partire dal giorno 1 gennaio 1970. Nello standard C, questo tipo derivato viene definito nel file '`time.h`'.
`uid_t`	Numero intero senza segno. Numero identificativo dell'utente a cui appartiene un file.

Tipo derivato	Descrizione
`id_t`	Numero intero senza segno. Numero identificativo generico, in grado di ospitare il valore dei tipi '**`pid_t`**', '**`uid_t`**' e '**`gid_t`**'. Va tenuto in considerazione il fatto che un numero di identificazione di un processo deve essere positivo, salvo il caso in cui un valore negativo serva per indicare un errore; pertanto, il tipo '**`id_t`**' può essere definito come privo di segno.

L'esempio successivo mostra come potrebbero essere dichiarati questi tipi derivati, limitatamente ai casi descritti nella tabella:

```
typedef long int           blkcnt_t;
typedef long int           blksize_t;
typedef long int           clock_t;
typedef unsigned long int  dev_t;
typedef unsigned int       id_t;
typedef unsigned long int  ino_t;
typedef unsigned int       gid_t;
typedef unsigned int       mode_t;
typedef unsigned int       nlink_t;
typedef long long int      off_t;
typedef int                pid_t;
typedef unsigned long int  pthread_t;
typedef unsigned long int  size_t;
typedef long int           ssize_t;
typedef long int           time_t;
typedef unsigned int       uid_t;
```

70.2 File «sys/stat.h»

Il file ‘`sys/stat.h`’ viene usato dallo standard POSIX principalmente per definire un insieme di macro-variabili che individuano le caratteristiche fondamentali di un file (tipo di file e permessi), per definire il tipo ‘**`struct stat`**’ che serve a rappresentare lo stato di un file, per dichiarare il prototipo di alcune funzioni che hanno a che fare con queste informazioni (si veda eventualmente la realizzazione del file ‘`sys/stat.h`’ e di alcune delle sue funzioni nei sorgenti di os32, sezione 95.25).

È importante considerare il file ‘`sys/stat.h`’ assieme a ‘`fcntl.h`’.

Nel file ‘`sys/stat.h`’ si fa riferimento a un insieme di tipi derivati, dichiarati nel file ‘`sys/types.h`’. Per semplicità, l’esempio propone la sua inclusione iniziale:

```
#include <sys/types.h>  // dev_t
                        // off_t
                        // blkcnt_t
                        // blksize_t
                        // ino_t
                        // mode_t
                        // nlink_t
                        // uid_t
                        // gid_t
                        // time_t
```

70.2.1 Macro-variabili per la definizione del contenuto di un valore di tipo «mode_t»

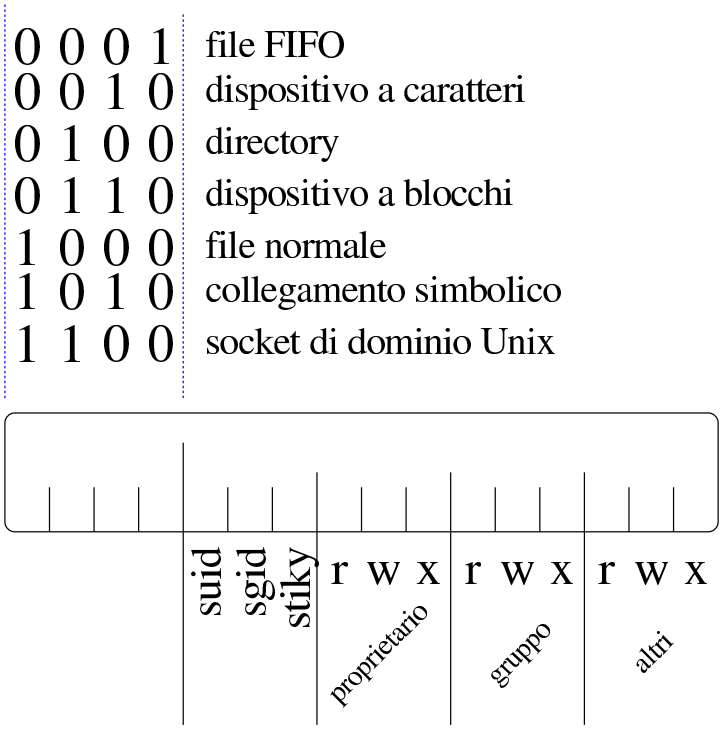
«

Il tipo derivato '**mode_t**' serve a rappresentare il tipo di un file (o di una directory) e i permessi disponibili. Questo tipo si traduce generalmente in un intero a 16 bit. Trattandosi di un valore numerico, queste informazioni sono distinte a gruppi di bit, selezionabili attraverso una maschera. Pertanto, tra le macro-variabili che distinguono le varie caratteristiche associabili a una variabile di tipo '**mode_t**', alcune vanno usate come maschera, per distinguere un certo insieme di informazioni, dalle altre.

```
//
// Tipo di file.
//
#define S_IFMT   0170000  // Maschera del tipo di file.
//
#define S_IFBLK  0060000  // File di dispositivo a blocchi.
#define S_IFCHR  0020000  // File di dispositivo a caratteri.
#define S_IFIFO  0010000  // File FIFO.
#define S_IFREG  0100000  // File puro e semplice
                          // (regular file).
#define S_IFDIR  0040000  // Directory.
#define S_IFLNK  0120000  // Collegamento simbolico.
#define S_IFSOCK 0140000  // Socket di dominio Unix.
//
// Permessi di accesso dell'utente proprietario del file.
//
#define S_IRWXU  0000700  // Maschera che rappresenta
                          // simultaneamente tutti i
                          // permessi per l'utente
                          // proprietario.
#define S_IRUSR  0000400  // Permesso di lettura.
#define S_IWUSR  0000200  // Permesso di scrittura.
```

```
#define S_IXUSR  0000100  // Permesso di esecuzione
                          // o attraversamento.
//
// Permessi di accesso del gruppo proprietario del file.
//
#define S_IRWXG  0000070  // Maschera che rappresenta
                          // simultaneamente tutti i
                          // permessi per il gruppo
                          // proprietario.
#define S_IRGRP  0000040  // Permesso di lettura.
#define S_IWGRP  0000020  // Permesso di scrittura.
#define S_IXGRP  0000010  // Permesso di esecuzione
                          // o attraversamento.
//
// Permessi di accesso degli altri utenti.
//
#define S_IRWXO  0000007  // Maschera che rappresenta
                          // simultaneamente tutti i
                          // permessi per il gruppo
                          // proprietario.
#define S_IROTH  0000004  // Permesso di lettura.
#define S_IWOTH  0000002  // Permesso di scrittura.
#define S_IXOTH  0000001  // Permesso di esecuzione
                          // o attraversamento.
//
// Permessi aggiuntivi: S-bit.
// In questo caso non c'è una maschera che li includa tutti.
//
#define S_ISUID  0004000  // S-UID.
#define S_ISGID  0002000  // S-GID.
#define S_ISVTX  0001000  // Sticky.
```

Figura 70.6. Schema delle «modalità» POSIX per descrivere le caratteristiche di un file. Questa informazione si associa all'inode.



Inoltre, vengono definite delle macroistruzioni per distinguere il tipo di file, dalle informazioni contenute in una variabile di tipo '**mode_t**', ammesso che queste informazioni siano incluse effettivamente:

```
//
// macroistruzioni per la verifica del tipo di file.
//
#define S_ISBLK(m)    (((m) & S_IFMT) == S_IFBLK)    // [1]
#define S_ISCHR(m)    (((m) & S_IFMT) == S_IFCHR)    // [2]
#define S_ISFIFO(m)   (((m) & S_IFMT) == S_IFIFO)    // [3]
#define S_ISREG(m)    (((m) & S_IFMT) == S_IFREG)    // [4]
#define S_ISDIR(m)    (((m) & S_IFMT) == S_IFDIR)    // [5]
#define S_ISLNK(m)    (((m) & S_IFMT) == S_IFLNK)    // [6]
#define S_ISSOCK(m)   (((m) & S_IFMT) == S_IFSOCK)   // [7]
// [1] Dispositivo a blocchi.
// [2] Dispositivo a caratteri.
```

```
// [3] File FIFO.
// [4] File puro e semplice.
// [5] È una directory.
// [6] Collegamento simbolico.
// [7] Socket di dominio Unix.
```

70.2.2 Struttura «stat»

Il file ‘sys/stat.h’ include la dichiarazione del tipo derivato ‘**struct stat**’, con lo scopo di contenere le informazioni disponibili su di un file. La struttura deve contenere almeno i membri che appaiono nell’esempio successivo:

```
//
// Struttura «stat».
//
struct stat {
    dev_t     st_dev;     // File di dispositivo contenente
                          // il file.
    ino_t     st_ino;     // Numero di serie del file
                          // (inode).
    mode_t    st_mode;    // Tipo e permessi del file.
    nlink_t   st_nlink;   // Collegamenti fisici associati
                          // al file.
    uid_t     st_uid;     // Numero identificativo
                          // dell'utente proprietario.
    gid_t     st_gid;     // Numero identificativo del
                          // gruppo proprietario.
    dev_t     st_rdev;    // Numero del file dispositivo, se
                          // si tratta di un tale tipo di
                          // file.
    off_t     st_size;    // Se si tratta di un file vero
                          // e proprio, misura la dimensione
                          // del file; se si tratta di un
```

```
                                // collegamento simbolico, misura
                                // la dimensione del percorso che
                                // questo rappresenta; per altri
                                // casi il significato di questo
                                // campo non è precisato.
    time_t    st_atime;         // Data dell'ultimo accesso.
    time_t    st_mtime;         // Data dell'ultima modifica del
                                // contenuto.
    time_t    st_ctime;         // Data dell'ultima modifica dello
                                // stato.
    blksize_t st_blksize;       // Dimensione ottimale del blocco
                                // per le operazioni di I/O.
    blkcnt_t  st_blocks;        // Blocchi da 512 byte allocati
                                // per il file.
};
```

70.2.3 Prototipi di funzione

«

Il file 'sys/stat.h' include la dichiarazione di alcuni prototipi di funzione, come si vede nell'esempio seguente:

```
//
// Prototipi di funzione.
//
int    chmod  (const char *path, mode_t mode);
int    fchmod (int fdn, mode_t mode);
int    fstat  (int fdn, struct stat *buf);
int    lstat  (const char *restrict file,
               struct stat *restrict buf);
int    mkdir  (const char *path, mode_t mode);
int    mkfifo (const char *path, mode_t mode);
int    mknod  (const char *path, mode_t mode, dev_t dev);
int    stat   (const char *restrict path,
               struct stat *restrict buf);
mode_t umask  (mode_t mask);
```

70.3 File «strings.h»

Il file di intestazione ‘`strings.h`’ contiene i prototipi di alcune funzioni, legate prevalentemente alla scansione delle stringhe. Dal momento che viene usato il tipo derivato ‘**`size_t`**’, questo viene definito attraverso l’inclusione del file ‘**`stddef.h`**’.

```
#include <stddef.h>

int ffs          (int i);
int strcasecmp   (const char *s1, const char *s2);
int strncasecmp  (const char *s1, const char *s2, size_t n);
```

Lo standard prevede anche altri prototipi di funzioni ormai superate, che rimangono solo per compatibilità con il passato.

70.3.1 Funzione «ffs()»

La funzione ***ffs()*** serve a scandire i bit di un valore numerico intero, alla ricerca del primo bit a uno, partendo dalla posizione meno significativa, restituendo l’indice di tale bit, considerando il bit meno significativo avente indice uno. Pertanto, se il valore da scandito è pari a zero (non ha alcun bit a uno), la funzione restituisce zero.

Al di fuori dello standard, è probabile trovare delle altre funzioni simili a questa, per la scansione degli interi di tipo ‘**`long int`**’ e di tipo ‘**`long long int`**’. In tal caso, i nomi delle funzioni ulteriori possono essere ***ffsl()*** e ***ffsll()***.

70.3.2 Funzioni «strcasecmp()» e «strncasecmp()»

«

Le funzioni ***strcasecmp()*** e ***strncasecmp()*** servono a confrontare due stringhe, ignorando la differenza tra maiuscole e minuscole. Nel caso di ***strncasecmp()*** il confronto è limitato a una certa quantità massima di caratteri.

Se la configurazione locale è quella POSIX, il confronto avviene come se le due stringhe venissero convertite preventivamente in caratteri minuscoli; ma nel caso sia attiva una configurazione locale differente, lo standard non specifica in che modo la comparazione abbia luogo.

Il valore restituito dalle due funzioni dipende dal confronto lessicografico delle due stringhe. Se sono uguali (a parte la differenza tra maiuscole e minuscole), il risultato è zero; se la prima stringa è lessicograficamente precedente rispetto alla seconda, il valore restituito è inferiore a zero; se la prima stringa è lessicograficamente successiva alla seconda, il valore ottenuto è superiore a zero.

70.4 File «fcntl.h»

«

Il file di intestazione ‘`fcntl.h`’ riguarda la parte fondamentale della gestione dei file, attraverso i descrittori; precisamente si considerano la creazione, l’apertura e l’attribuzione di opzioni di funzionamento, mentre altre questioni sono gestite attraverso le definizioni contenute nel file ‘`unistd.h`’ (si veda eventualmente la realizzazione del file ‘`fcntl.h`’ e di alcune delle sue funzioni nei sorgenti di os32, sezione 95.6).

70.4.1 Tipi di dati derivati

Il file di intestazione ‘`fcntl.h`’ utilizza alcuni tipi di dati derivati, già definiti nel file ‘`sys/types.h`’:

```
#include <sys/types.h>  // mode_t
                        // off_t
                        // pid_t
```

70.4.2 Opzioni per la funzione «fcntl()»

Nel file di intestazione ‘`fcntl.h`’ si definiscono varie macro-variabili, di cui un primo insieme riguarda, quasi in modo esclusivo, l’uso della funzione ***fcntl()***.

```
//
// Valori per il secondo argomento della funzione fcntl().
//
#define F_DUPFD         0 // Duplicate file descriptor.
#define F_GETFD         1 // Get file descriptor flags.
#define F_SETFD         2 // Set file descriptor flags.
#define F_GETFL         3 // Get file status flags.
#define F_SETFL         4 // Set file status flags.
#define F_GETLK         5 // Get record locking information.
#define F_SETLK         6 // Set record locking information.
#define F_SETLKW        7 // Set record locking information;
                          // wait if blocked.
#define F_GETOWN        8 // Set owner of socket.
#define F_SETOWN        9 // Get owner of socket.
//
// Flag da impostare con:
//   fcntl (fd, F_SETFD, ...);
//
#define FD_CLOEXEC      1 // Chiude il descrittore del file
```

```
                                    // nel momento dell'esecuzione di
                                    // una funzione del gruppo
                                    // 'exec()'.
```

Le macro-variabili ***F_DUPFD***, ***F_GETFD***, ***F_SETFD***, ***F_GETFL***, ***F_SETFL***, ***F_GETLK***, ***F_SETLK***, ***F_SETLKW***, ***F_GETOWN*** e ***F_SETOWN***, servono per dichiarare il tipo di comando da dare alla funzione ***fcntl()***, attraverso il suo secondo parametro:

```
int fcntl (int fdn, int cmd, ...);
```

La macro-variabile ***FD_CLOEXEC*** riguarderebbe un insieme di indicatori associati a un descrittore di file (***fd_flags***), di cui però ne esiste uno solo, rappresentato dalla macro-variabile stessa. Utilizzando la funzione ***fcntl()*** e specificando il comando ***F_GETFD*** è possibile ottenere lo stato di questi indicatori (ovvero solo ***FD_CLOEXEC***), mentre con il comando ***F_SETFD*** è possibile modificare questo stato. Quando l'indicatore ***FD_CLOEXEC*** risulta attivo per un certo descrittore di file, se viene eseguita la sostituzione del processo con l'ausilio di una funzione del gruppo ***exec**...()***, il descrittore in questione viene chiuso, mentre diversamente rimarrebbe aperto.

Nel file di intestazione '`fcntl.h`' vengono dichiarate anche delle macro-variabili per definire la modalità di accesso a un file, da usare prevalentemente con la funzione ***open()***:

```
//
// Indicatori per la creazione dei file, da usare nel
// parametro «oflag» della funzione open().
//
#define O_CREAT    000010 // Crea il file se non esiste già.
```

```
#define O_EXCL      000020 // Indicatore di accesso
                           // esclusivo.
#define O_NOCTTY    000040 // Non assegna un terminale di
                           // controllo.
#define O_TRUNC     000100 // Indicatore di troncamento.
//
// Indicatori dello stato dei file,
// usati nelle funzioni open() e fcntl().
//
#define O_APPEND    000200 // Scrittura in aggiunta.
#define O_DSYNC     000400 // Scrittura sincronizzata
                           // dei dati.
#define O_NONBLOCK  001000 // Modalità non bloccante.
#define O_RSYNC     002000 // Lettura sincronizzata.
#define O_SYNC      004000 // Scrittura sincronizzata
                           // dei file.
//
// Maschera per la selezione delle sole modalità principali
// di accesso ai file.
//
#define O_ACCMODE   000003 // Seleziona gli ultimi due bit,
                           // che in questo caso individuano
                           // le modalità di accesso
                           // principali (lettura, scrittura
                           // ed entrambe) dalle altre
                           // modalità che sono già state
                           // descritte sopra.
//
// Modalità principali di accesso ai file, secondo la
// tradizione.
//
//#define O_RDONLY 000000 // Apertura in sola lettura.
//#define O_WRONLY 000001 // Apertura in sola scrittura.
//#define O_RDWR   000002 // Apertura in lettura e
```

```
//                              // scrittura.
//
// Modalità principali secondo le indicazioni per le
// realizzazioni future:
//
#define O_RDONLY    000001 // Apertura in lettura.
#define O_WRONLY    000002 // Apertura in scrittura.
#define O_RDWR      (O_RDONLY | O_WRONLY) // Apertura in
                                          // lettura
                                          // e scrittura.
```

70.4.3 Gestione del blocco dei file

«

Alcune macro-variabili e il tipo derivato '**`struct flock`**' servono per la gestione del blocco dei file.[1] L'esempio seguente si limita ai soli membri indispensabili:

```
//
// Valori per il tipo "l_type" (di una variabile strutturata
// di tipo "struct flock"), usato per il blocco dei record
// attraverso la funzione fcntl().
//
#define F_RDLCK         0       // Read lock.
#define F_WRLCK         1       // Write lock.
#define F_UNLCK         2       // Remove lock.
//
// Struttura «flock».
//
struct flock {
    short int l_type;   // Tipo del blocco: F_RDLCK,
                        // F_WRLCK, o F_UNLCK.
    short int l_whence; // Punto di riferimento iniziale a
                        // cui si somma poi il valore dello
                        // scostamento, rappresentato
```

```
                        // da "l_start".
    off_t     l_start;  // Scostamento che individua
                        // l'inizio dell'area bloccata.
                        // Lo scostamento può essere
                        // positivo o negativo.
    off_t     l_len;    // Dimensione dell'area bloccata:
                        // se si indica zero, significa che
                        // questa raggiunge la fine
                        // del file.
    pid_t     l_pid;    // Il numero del processo
                        // elaborativo che blocca
                        // l'area.
};
```

70.4.4 Funzioni

Sono presenti anche i prototipi delle funzioni ***creat()***, ***fcntl()*** e ***open()***:

```
//
// Prototipi di funzione.
//
int creat (const char *file, mode_t mode);
int fcntl (int fdn, int cmd, ...);
int open  (const char *file, int oflag, ...);
```

Per l'uso delle funzioni ***open()*** e ***creat()*** si veda la sezione 68.5.

La funzione ***fcntl()*** esegue un'operazione, definita dal parametro ***cmd***, sul descrittore richiesto come primo parametro (***fdn***). A seconda del tipo di operazione richiesta, potrebbero essere necessari degli argomenti ulteriori, i quali però non possono essere formalizzati in modo esatto nel prototipo della funzione. Il valore del secondo pa-

rametro che rappresenta l'operazione richiesta, va fornito in forma di costante simbolica, come descritto nell'elenco seguente, nel quale però sono descritti solo alcuni dei comandi possibili.

Sintassi	Descrizione
`fcntl (`***fdn***`, F_DUPFD, (int)` ***fdn_min***`)`	Richiede la duplicazione del descrittore di file ***fdn***, in modo tale che la copia abbia il numero di descrittore minore possibile, ma maggiore o uguale a quello indicato come argomento ***fdn_min***.
`fcntl (`***fdn***`, F_GETFD)` `fcntl (`***fdn***`, F_SETFD, (int)` ***fd_flags***`)`	Rispettivamente, legge o imposta, gli indicatori del descrittore di file ***fdn*** (eventualmente noti come *file descriptor flags* o solo *fd flags*). Per il momento, è possibile impostare un solo indicatore, ***FD_CLOEXEC***, pertanto, al posto di ***fd_flags*** si può mettere solo la costante ***FD_CLOEXEC***.
`fcntl (`***fdn***`, F_GETFL)` `fcntl (`***fdn***`, F_SETFL, (int)` ***fl_flags***`)`	Rispettivamente, legge o imposta, gli indicatori dello stato del file, relativi al descrittore ***fdn*** (eventualmente noti come *file flags* o solo *fl flags*). Per impostare questi indicatori, vanno combinate delle costanti simboliche: ***O_RDONLY***, ***O_WRONLY***, ***O_RDWR***, ***O_CREAT***, ***O_EXCL***, ***O_NOCTTY***, ***O_TRUNC***.

Il significato del valore restituito dalla funzione dipende dal tipo di operazione richiesta, come sintetizzato dalla tabella successiva, relativa ai soli comandi già apparsi. In generale, anche per gli altri comandi, un risultato erroneo viene comunque evidenziato dalla restituzione di un valore negativo.

Operazione richiesta	Significato del valore restituito
`F_DUPFD`	Si ottiene il numero del descrittore prodotto dalla copia, oppure −1 in caso di errore.
`F_GETFD`	Si ottiene il valore degli indicatori del descrittore (*fd flags*), oppure −1 in caso di errore.
`F_GETFL`	Si ottiene il valore degli indicatori del file (*fl flags*), oppure −1 in caso di errore.

70.5 File «unistd.h»

Il file di intestazione ‘`unistd.h`’ raccoglie un po’ di tutto ciò che riguarda le estensioni POSIX, pertanto è frequente il suo utilizzo (si veda eventualmente la realizzazione del file ‘`unistd.h`’ e di alcune delle sue funzioni nei sorgenti di os32, sezione 95.30).

Nel file ‘`unistd.h`’ si distingue la presenza di un elenco numeroso di macro-variabili con prefissi ***_POSIX_***…, ***_POSIX2_***… e ***_XOPEN_***…, con lo scopo di dichiarare le caratteristiche del sistema e della libreria. Per l’interrogazione delle caratteristiche o delle limitazioni del sistema, si utilizzano però delle funzioni apposite, costituite precisamente da ***pathconf()*** e ***sysconf()***, le quali utilizzano un proprio insieme di macro-variabili per individuare le caratteristiche da interrogare. Nel caso di ***pathconf()*** si aggiungono macro-variabili con prefisso ***_PC_***…; per ***sysconf()*** le macro-variabili hanno il prefisso ***_SC_***….

Nei prototipi di funzione si utilizzano diversi tipi derivati, già dichiarati nei file ‘sys/types.h’ e ‘inttypes.h’:

```
#include <sys/types.h>   // size_t, ssize_t, uid_t, gid_t,
                         // off_t, pid_t, useconds_t
#include <inttypes.h>    // intptr_t
```

Nel file deve anche essere dichiarato il valore per la macro-variabile ***NULL***; in questo caso viene incorporato il file ‘stddef.h’:

```
#include <stddef.h>      // NULL
```

70.5.1 Denominazione dei descrittori di file

«

Per dare un nome ai descrittori dei flussi standard, nel file ‘unistd.h’ si dichiarano tre macro-variabili, il cui valore è stabilito necessariamente:

```
#define STDIN_FILENO    0
#define STDOUT_FILENO   1
#define STDERR_FILENO   2
```

70.5.2 Verifica dei permessi di accesso

«

La funzione ***access()*** consente di verificare l’accessibilità di un file. Per questo richiede l’indicazione del percorso e di un valore che rappresenta i tipi di accesso che si vogliono considerare. Questi sono rappresentati dall’unione di quattro possibili macro-variabili:

```
#define R_OK    04   // Accessibilità in lettura.
#define W_OK    02   // Accessibilità in scrittura.
#define X_OK    01   // Accessibilità in esecuzione o
                     // attraversamento.
#define F_OK    00   // Esistenza del file.
```

I valori che questa macro-variabili possono avere devono essere tali da consentire la combinazione con l'operatore '|' (OR, bit per bit), così da poter verificare simultaneamente tutti gli aspetti dell'accesso al file.

70.5.3 Funzioni

Segue l'elenco dei prototipi delle funzioni principali del file 'unistd.h':

```
int           access      (const char *path, int mode);
unsigned int  alarm       (unsigned int seconds);
int           chdir       (const char *path);
int           chown       (const char *path, uid_t owner,
                           gid_t group);
int           close       (int fdn);
size_t        confstr     (int name, char *buffer,
                           size_t length);
int           dup         (int old_fdn);
int           dup2        (int old_fdn, int new_fdn);
int           execl       (const char *path,
                           const char *arg, ...);
int           execle      (const char *path,
                           const char *arg, ...,
                           char *const envp[]);
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 owner,
                            gid_t group);
pid_t         fork         (void);
long          fpathconf    (int fdn, int name);
int           ftruncate    (int fdn, off_t length);
char         *getcwd       (char *buffer, size_t size);
gid_t         getegid      (void);
uid_t         geteuid      (void);
gid_t         getgid       (void);
int           getgroups    (int size, gid_t list[]);
int           gethostname  (char *name, size_t length);
char         *getlogin     (void);
int           getlogin_r   (char *buffer, size_t size);
int           getopt       (int argv, char *const argv[],
                            const char *optstring);
pid_t         getpgrp      (void);
pid_t         getpid       (void);
pid_t         getppid      (void);
uid_t         getuid       (void);
int           isatty       (int fdn);
int           link         (const char *old_path,
                            const char *new_path);
off_t         lseek        (int fdn, off_t offset,
                            int whence);
long          pathconf     (const char *path, int name);
int           pause        (void);
int           pipe         (int fdn[2]);
ssize_t       read         (int fdn, void *buffer,
                            size_t length);
ssize_t       readlink     (const char *restrict path,
                            char *restrict buffer,
                            size_t size);
int           rmdir        (const char *path);
```

```
int             setegid     (gid_t egid);
int             seteuid     (uid_t euid);
int             setgid      (gid_t gid);
int             setpgid     (pid_t pid, pid_t pgid);
pid_t           setsid      (void);
int             setuid      (uid_t uid);
unsigned        sleep       (unsigned int seconds);
int             symlink     (const char *old_path,
                             const char *new_path);
long            sysconf     (int name);
pid_t           tcgetpgrp   (int fdn);
int             tcsetpgrp   (int fdn, pid_t pgrp);
char           *ttyname     (int fdn);
int             ttyname_r   (int fdn, char *buffer,
                             size_t length);
int             unlink      (const char *path);
ssize_t         write       (int fdn, const void *buffer,
                             size_t length);
```

70.5.4 Funzioni di servizio per la gestione di file e directory

Nelle sezioni seguenti si descrivono solo alcune delle funzioni destinate alla gestione di file e directory, il cui prototipo appare nel file ‘`unistd.h`’.

70.5.4.1 Funzione «access()»

La funzione ***access()*** consente di verificare l’accessibilità di un file, il cui percorso viene fornito tramite una stringa. L’accessibilità viene valutata in base a delle opzioni, con cui è possibile specificare a cosa si è interessati in modo preciso.

```
int access (const char *percorso, int modalità);
```

Il secondo parametro della funzione è un numero intero fornito normalmente attraverso l'indicazione di una macro-variabile che rappresenta simbolicamente il tipo di accesso che si intende verificare. Si può utilizzare ***F_OK*** per verificare l'esistenza del file o della directory; in alternativa, si possono usare le macro-variabili ***R_OK***, ***W_OK*** e ***X_OK***, sommabili assieme attraverso l'operatore OR binario, per la verifica dell'accessibilità in lettura, in scrittura e in esecuzione o attraversamento. Per esempio, scrivendo '**`R_OK|W_OK|X_OK`**' si vuole verificare che il file o la directory sia accessibile con tutti i tre permessi attivi.

Tabella 70.23. Macro-variabili usate per descrivere la modalità di accesso del secondo parametro della funzione ***access()***.

Macro-variabile	Descrizione
`F_OK`	Si richiede la verifica dell'esistenza del file o della directory. Questa opzione va usata da sola e non può essere sommata alle altre.
`R_OK`	Si richiede la verifica dell'accessibilità in lettura del file o della directory.
`W_OK`	Si richiede la verifica dell'accessibilità in scrittura del file o della directory.
`X_OK`	Si richiede la verifica dell'accessibilità in esecuzione del file o in attraversamento della directory.

La funzione restituisce il valore zero se il file o la directory risultano accessibili nel modo richiesto, altrimenti restituisce il valore −1 e si può verificare il tipo di errore valutando il contenuto della variabile

errno.

L'esempio seguente descrive completamente l'uso della funzione. Si può osservare che per valutare il successo dell'operazione, l'esito restituito dalla funzione viene invertito; inoltre, il contenuto della variabile ***errno*** viene considerato con l'aiuto della funzione ***perror()***. Il file dell'esempio dovrebbe essere disponibile presso *allegati/c/esempio-posix-unistd-access.c* .

```
#include <stdio.h>
#include <unistd.h>

int
main (void)
{
    const char *file_name = "/tmp/test";

    if (! access (file_name, F_OK))
      {
        printf ("Il file o la directory "
                "\"%s\" esiste.\n", file_name);
        if (! access (file_name, R_OK))
          {
            printf ("Il file o la directory "
                    "\"%s\" ", file_name);
            printf ("è accessibile in lettura.\n");
          }
        else
          {
            printf ("Il file o la directory "
                    "\"%s\" ", file_name);
            printf ("non è accessibile in lettura.\n");
            perror (NULL);
          }
        if (! access (file_name, W_OK))
```

```
          {
            printf ("Il file o la directory "
                    "\"%s\" ", file_name);
            printf ("è accessibile in scrittura.\n");
          }
        else
          {
            printf ("Il file o la directory "
                    "\"%s\" ", file_name);
            printf ("non è accessibile in scrittura.\n");
            perror (NULL);
          }
        if (! access (file_name, X_OK))
          {
            printf ("Il file o la directory "
                    "\"%s\" ", file_name);
            printf ("è accessibile in esecuzione o "
                    "attraversamento.\n");
          }
        else
          {
            printf ("Il file o la directory "
                    "\"%s\" ", file_name);
            printf ("non è accessibile in esecuzione "
                    "o attraversamento.\n");
            perror (NULL);
          }
      }
    else
      {
        printf ("Il file o la directory "
                "\"%s\" non esiste.\n", file_name);
        perror (NULL);
      }
```

```
    return (0);
}
```

70.5.4.2 Funzione «getcwd()»

La funzione ***getcwd()*** (*get current working directory*) consente di annotare in una stringa il percorso della directory corrente.

```
char *getcwd (char *buffer, size_t max);
```

Come si può vedere dal prototipo della funzione, occorre predisporre prima un array di caratteri, da usare come stringa, in cui la funzione va a scrivere il percorso trovato (con tanto di carattere nullo di terminazione). Come secondo parametro della funzione va indicata la dimensione massima dell'array, oltre la quale la scrittura non può andare.

La funzione restituisce il puntatore alla stringa contenente il percorso, ma se si verifica un errore restituisce il puntatore nullo e aggiorna la variabile ***errno*** con la specificazione della causa di tale errore.

L'esempio seguente descrive l'uso della funzione in modo molto semplice; in particolare, in caso di errore si usa la funzione ***perror()*** per visualizzarne una descrizione. Il file dell'esempio dovrebbe essere disponibile presso *allegati/c/esempio-posix-unistd-getcwd.c* .

```
#include <stdio.h>
#include <unistd.h>

int
main (void)
```

```
{
    char cwd[500];
    char *result;

    result = getcwd (cwd, 500);

    if (result == NULL)
      {
        perror (NULL);
      }
    else
      {
        printf ("%s\n", cwd);
      }
    return (0);
}
```

70.5.4.3 Funzione «chdir()»

La funzione ***chdir()*** (*change directory*) consente di cambiare la directory corrente del processo elaborativo.

```
int chdir (const char *path);
```

La funzione richiede come unico parametro la stringa che descrive il percorso da raggiungere. La funzione restituisce zero se l'operazione si conclude con successo, oppure il valore −1 in caso di errore, ma in tal caso viene modificata anche la variabile ***errno*** con l'indicazione più precisa dell'errore verificatosi.

L'esempio seguente mostra il comportamento della funzione, ma va osservato che l'effetto riguarda esclusivamente il processo in fun-

zione e non si riflette al processo che lo genera a sua volta. Per questa ragione il programma di esempio visualizza la posizione corrente raggiunta. Il file dell'esempio dovrebbe essere disponibile presso *allegati/c/esempio-posix-unistd-chdir.c* .

```
#include <stdio.h>
#include <unistd.h>

int
main (void)
{
    const char *path = "/tmp/test";
    char cwd[500] = {'\0'};

    if (! chdir (path))
      {
        getcwd (cwd, 500);
        printf ("Il processo passa alla "
                "directory \"%s\".\n", cwd);
      }
    else
      {
        printf ("Non è possibile passare alla "
                "directory \"%s\"!\n",
                path);
        perror (NULL);
      }

    return (0);
}
```

70.5.4.4 Funzioni «link()» e «unlink()»

«

La funzione ***link()*** consente la creazione di un nuovo collegamento fisico a partire da un file o da una directory già esistente, tenendo conto che la facoltà di creare un collegamento fisico a partire da una directory è una funzione privilegiata e il sistema operativo potrebbe non ammetterla in ogni caso. Per collegamento fisico si intende il riferimento contenuto in una directory, verso un certo file o una certa sottodirectory individuati numericamente da un numero inode. La funzione ***link()*** produce una sorta di copia del file, nel senso che si predispone un riferimento aggiuntivo alla stessa cosa, senza la duplicazione dei dati relativi. Per converso, la funzione ***unlink()*** elimina il riferimento a un file o a una directory, cosa che coincide con la cancellazione del file o della directory, se si tratta dell'ultimo riferimento esistente a tale oggetto nel file system. Anche in questo caso, va considerato in modo particolare l'eliminazione del riferimento a una directory: il sistema operativo può impedirlo se si tratta di una directory non vuota.

```
int link (const char *p1, const char *p2);
```

```
int unlink (const char *p);
```

I parametri delle due funzioni sono stringhe che descrivono il percorso di un file o di una directory. Nel caso di ***unlink()*** si indica solo il percorso da rimuovere, mentre con ***link()*** se ne indicano due: l'origine e la destinazione che si vuole creare.

Le due funzioni restituiscono un valore intero pari a zero se tutto

va bene, altrimenti restituiscono il valore −1, modificando anche il contenuto della variabile ***errno*** con un'informazione più precisa sull'accaduto.

Gli esempi seguenti mostrano il comportamento delle due funzioni. I file degli esempi dovrebbero essere disponibili presso *allegati/c/esempio-posix-unistd-link.c* e *allegati/c/esempio-posix-unistd-unlink.c* .

```
#include <stdio.h>
#include <unistd.h>

int
main (void)
{
    const char *old = "/tmp/test";
    const char *new = "/tmp/test.link";

    if (! link (old, new))
      {
        printf ("Creato il collegamento \"%s\".\n", new);
      }
    else
      {
        printf ("Non è possibile creare il "
                "collegamento \"%s\"!\n",
                new);
        perror (NULL);
      }

    return (0);
}
```

```
#include <stdio.h>
#include <unistd.h>
```

```
int
main (void)
{
    const char *name = "/tmp/test";

    if (! unlink (name))
      {
        printf ("Cancellato il file o la "
                "directory \"%s\".\n", name);
      }
    else
      {
        printf ("Non è possibile cancellare il file "
                "o la directory ");
        printf ("\"%s\"!\n", name);
        perror (NULL);
      }

    return (0);
}
```

70.6 File «dirent.h»

«

Il file di intestazione ‘`dirent.h`’ raccoglie ciò che serve per la gestione delle directory, attraverso flussi individuati da puntatori di tipo ‘**`DIR *`**’ (si veda eventualmente la realizzazione del file ‘`dirent.h`’ e di alcune delle sue funzioni nei sorgenti di os32, sezione 95.4). La gestione di tali flussi può avvenire attraverso i descrittori di file, così come già avviene per i flussi individuati da puntatori di tipo ‘**`FILE *`**’, ma ciò è una facoltà, non una necessità realizzativa.

70.6.1 Struttura «dirent»

Il file ‘`dirent.h`’ prevede la dichiarazione di un tipo derivato, denominato ‘**`struct dirent`**’, da usare per rappresentare una voce singola di una directory. I membri necessari di questa struttura sono ***d_ino***, per rappresentare il numero di inode della voce, e ***d_name[]*** per contenere il nome del file relativo, completo di carattere nullo di terminazione delle stringhe. La definizione del tipo ‘**`struct dirent`**’ potrebbe essere realizzata nel modo seguente:

```
#include <sys/types.h>  // ino_t
#include <limits.h>     // NAME_MAX
//
struct dirent {
    ino_t  d_ino;                   // Numero di inode
    char   d_name[NAME_MAX+1];      // NAME_MAX + '\0'
} __attribute__ ((packed));
```

70.6.2 Tipo «DIR»

Nel file ‘`dirent.h`’ è definito il tipo derivato ‘**`DIR`**’, con il quale si intende fare riferimento a un flusso che individua una directory aperta. Se la gestione di tali flussi avviene attraverso i descrittori di file, ci deve poter essere il riferimento al numero del descrittore relativo. Quello che segue è un esempio di una tale struttura, seguita dalla dichiarazione di un array per il contenimento delle informazioni su tutte le directory aperte del processo:

```
typedef struct {
    int            fdn;     // Numero del descrittore di file.
    struct dirent dir;      // Last directory item read.
} DIR;

extern DIR _directory_stream[]; // Array di directory che
                                // però non è previsto
                                // espressamente dallo
                                // standard.
```

Nella struttura di tipo '**DIR**' di questo esempio, viene inclusa una struttura di tipo '**struct dirent**', per permettere alla funzione ***readdir()*** di annotare l'ultima voce letta da una certa directory.

70.6.3 Prototipi di funzioni

«

La gestione delle directory in forma di flussi di tipo '**DIR ***' richiede alcune funzioni specifiche, di cui si trovano i prototipi nel file 'dirent.h':

```
int            closedir  (DIR *dp);
DIR           *opendir   (const char *name);
struct dirent *readdir   (DIR *dp);
void           rewinddir (DIR *dp);
```

La funzione ***opendir()*** apre un flusso corrispondente a una directory indicata tramite il suo percorso, restituendo il puntatore relativo. Una volta aperta una directory, si possono leggere le sue voci con la funzione ***readdir()***, la quale restituisce il puntatore di una variabile strutturata di tipo '**struct dirent**', all'interno della quale è possibile trarre i dati della voce letta: negli esempi di questo capitolo, tali informazioni sono incorporate nella struttura '**DIR**' che rappresenta la directory aperta. Ogni lettura fa avanzare alla voce successiva della directory e, se necessario, si può riposizionare l'indice di lettura

alla prima voce, con la funzione ***rewinddir()***. Per chiudere un flusso aperto, si usa la funzione ***closedir()***.

70.7 File «termios.h»

La gestione essenziale del terminale a caratteri è abbastanza complessa e si sintetizza con le definizioni del file ‘`termios.h`’ (si veda eventualmente la realizzazione del file ‘`termios.h`’ e di alcune delle sue funzioni nei sorgenti di os32, sezione 95.28). Lo standard prevede due modalità di inserimento: canonica e non canonica. Negli esempi di questo capitolo ci si sofferma su ciò che serve nel file ‘`termios.h`’ per gestire la modalità canonica, ovvero quella tradizionale per cui il dispositivo del terminale fornisce dati, solo dopo l’inserimento completo di una riga, confermato con un codice di interruzione di riga o con un altro codice che concluda comunque l’inserimento.

70.7.1 Tipi speciali

Nel file ‘`termios.h`’ vengono definiti alcuni tipi speciali per variabili scalari, che potrebbero essere descritti nel modo seguente:

```
typedef unsigned int tcflag_t;
typedef unsigned int speed_t;
typedef unsigned int cc_t;
```

Il tipo ‘**`tcflag_t`**’ viene usato nelle strutture di tipo ‘**`struct termios`**’, la cui definizione viene fatta nello stesso file ‘`termios.h`’; il tipo ‘**`speed_t`**’ serve per contenere il valore di una velocità di comunicazione del terminale; il tipo ‘**`cc_t`**’ serve per rappresentare un carattere di controllo, per la gestione del terminale.

70.7.2 Tipo «struct termios»

«

Nel file ‘`termios.h`’ viene definita la struttura ‘**`struct termios`**’, allo scopo di annotare tutte le informazioni sulla modalità di funzionamento di un certo dispositivo di terminale. Nell’esempio seguente si vedono solo i membri obbligatori, ma va considerata l’aggiunta di informazioni legate alla velocità di comunicazione, se il terminale ne deve fare uso. La definizione della struttura ‘**`struct termios`**’ richiede anche la dichiarazione della macro-variabile ***NCCS***, come si vede nell’esempio.

```
#define NCCS    11      // Dimensione dell'array 'c_cc[]'.
//
struct termios {
    tcflag_t c_iflag;
    tcflag_t c_oflag;
    tcflag_t c_cflag;
    tcflag_t c_lflag;
    cc_t     c_cc[NCCS];
};
```

Il membro ***c_iflag*** contiene gli indicatori che descrivono la modalità di inserimento di dati attraverso il terminale; il membro ***c_oflag*** contiene indicatori per la modalità di elaborazione dei dati in uscita, prima della loro lettura; il membro ***c_cflag*** contiene opzioni di controllo; il membro ***lflag*** contiene opzioni «locali» (qui, in particolare, si definisce se il terminale debba funzionare in modalità canonica o meno); l’array ***c_cc[]*** contiene i codici di caratteri da interpretare in modo speciale, per il controllo del funzionamento del terminale.

70.7.3 Codici di controllo

Durante l'inserimento di dati attraverso il terminale, alcuni codici possono assumere un significato particolare. Il valore numerico di questi codici è annotato nell'array ***c_cc[]*** che è membro della struttura di tipo '**struct termios**'; per farvi riferimento, l'indice da usare nell'array ***c_cc[]*** deve essere indicato attraverso una meta-variabile:

```
#define VEOF      0     // carattere EOF
#define VEOL      1     // carattere EOL
#define VERASE    2     // carattere ERASE
#define VINTR     3     // carattere INTR
#define VKILL     4     // carattere KILL
#define VMIN      5     // valore    MIN
#define VQUIT     6     // carattere QUIT
#define VSTART    7     // carattere START
#define VSTOP     8     // carattere STOP
#define VSUSP     9     // carattere SUSP
#define VTIME    10     // valore    TIME
```

Due valori di questo elenco fanno eccezione: '**VMIN**' e '**VTIME**', in quanto rappresentano invece un'informazione di tipo differente, necessaria per la gestione non canonica del terminale.

70.7.4 Indicatori per il membro «c_iflag»

Il membro ***c_iflag*** di una struttura di tipo '**struct termios**' può contenere degli indicatori indipendenti sulla configurazione per l'inserimento. Tali indicatori vanno forniti attraverso macro-variabili definite nel file '`termios.h`', di cui segue un esempio:

```
#define BRKINT    1 // Invia un segnale di interruzione
                    // in caso di ricevimento di un
                    // carattere INTR.
#define ICRNL     2 // Converte <CR> in <NL>.
#define IGNBRK    4 // Ignora il carattere INTR.
#define IGNCR     8 // Ignora il carattere <CR>.
#define IGNPAR   16 // Ignora i caratteri con errori di
                    // parità.
#define INLCR    32 // Converte <NL> in <CR>.
#define INPCK    64 // Abilita il controllo di parità.
#define ISTRIP  128 // Azzera l'ottavo bit dei caratteri.
#define IXOFF   256 // Abilita il controllo start/stop in
                    // ingresso.
#define IXON    512 // Abilita il controllo start/stop in
                    // uscita.
#define PARMRK 1024 // Segnala gli errori di parità.
```

70.7.5 Indicatori per il membro «c_oflag»

«

Il membro ***c_oflag*** di una struttura di tipo '**struct termios**' può contenere degli indicatori indipendenti sulla configurazione per l'output. Tali indicatori vanno forniti attraverso macro-variabili definite nel file '`termios.h`'. Di questi indicatori ne è obbligatorio solo uno: ***OPOST***, che ha lo scopo di abilitare una forma imprecisata di elaborazione dell'output.

```
#define OPOST     1     // post-process output
```

70.7.6 Indicatori per il membro «c_cflag»

«

Il membro ***c_cflag*** di una struttura di tipo '**struct termios**' può contenere degli indicatori indipendenti sulla configurazione per il controllo del terminale. Tale configurazione qui viene omessa.

70.7.7 Indicatori per il membro «c_lflag»

Il membro ***c_lflag*** di una struttura di tipo '**`struct termios`**' può contenere degli indicatori indipendenti sulla configurazione «locale» del terminale. Tali indicatori vanno forniti attraverso macro-variabili definite nel file '`termios.h`', di cui segue un esempio:

```
#define ECHO      1 // Abilita l'eco del terminale.
#define ECHOE     2 // Visualizza la cancellazione di un
                    // carattere.
#define ECHOK     4 // Visualizza l'eliminazione di una
                    // riga.
#define ECHONL    8 // Visualizza l'effetto del codice di
                    // interruzione di riga.
#define ICANON   16 // Modalità di inserimento canonica.
#define IEXTEN   32 // Modalità di inserimento estesa
                    // (non canonica).
#define ISIG     64 // Abilita i segnali.
#define NOFLSH  128 // Disabilita lo scarico della memoria
                    // tampone dopo un'interruzione o una
                    // conclusione forzata.
#define TOSTOP  256 // Invia il segnale SIGTTOU per l'output
                    // sullo sfondo.
```

L'indicatore ***ICANON*** consente di ottenere un funzionamento del terminale in modalità canonica.

70.7.8 Funzioni

Quanto descritto fino a questo punto sul file '`termios.h`' è ciò che poi serve per l'uso di alcune funzioni, il cui scopo è quello di configurare o interrogare la configurazione di un terminale. Nell'esempio seguente appaiono solo alcuni prototipi, assieme alla dichiarazione di alcune macro-variabili necessarie per la funzione ***tcsetattr()***.

```
#define TCSANOW   1 // Cambia gli attributi immediatamente.
#define TCSADRAIN 2 // Cambia gli attributi quando l'output
                    // è stato utilizzato.
#define TCSAFLUSH 3 // Cambia gli attributi quando l'output
                    // è stato utilizzato e scarica l'input
                    // ancora sospeso.

int tcgetattr (int fdn, struct termios *termios_p);
int tcsetattr (int fdn, int action,
               struct termios *termios_p);
```

Le due funzioni mostrate nell'esempio richiedono l'indicazione del numero di descrittore associato a un terminale aperto. Il parametro ***termios_p*** è un puntatore a una struttura con le informazioni sulla configurazione del terminale: la funzione ***tcgetattr()*** serve a ottenere la configurazione attuale, mentre la funzione ***tcsetattr()*** serve a modificarla. Il parametro ***action*** di ***tcsetattr()*** richiede di precisare la tempestività di tale modifica attraverso una delle macro-variabili elencate poco sopra.

70.8 Riferimenti

«

- Wikipedia, *C POSIX library*, *http://en.wikipedia.org/wiki/C_POSIX_library*
- The Open Group, *The Single UNIX® Specification, Version 2, System Interface & Headers Issue 5*, *http://pubs.opengroup.org/onlinepubs/007908799/xshix.html*
- Free Software Foundation, *The GNU C Library*, *http://www.gnu.org/software/libc/manual/*
- The Open Group, *The Single UNIX® Specification, Version 2, sys/types.h, sys/stat.h strings.h fcntl.h unistd.h dirent.h termios.h*

http://pubs.opengroup.org/onlinepubs/000095399/basedefs/sys/types.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/sys/stat.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/strings.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/fcntl.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/unistd.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/dirent.h.html ,
http://pubs.opengroup.org/onlinepubs/000095399/basedefs/termios.h.html

- Pagina di manuale *fcntl(2)*
- The Open Group, *The Single UNIX® Specification, Version 2*, *fcntl*, *http://pubs.opengroup.org/onlinepubs/000095399/functions/fcntl.html*
- The Open Group, *The Single UNIX® Specification, Version 2*, *General Terminal Interface*, *http://pubs.opengroup.org/onlinepubs/000095399/basedefs/xbd_chap11.html*

[1] Di norma, il programmatore non accede direttamente a variabili di tipo '**`struct flock`**', perché per la gestione dei blocchi si usano semplicemente le funzioni appropriate.